

# APPENDIX C

## COUNTERACTING GEOMETRIC DISTORTIONS IN WATERMARKING

### ALIGN.CPP

```

//*****
// FILE: Align.cpp
//*****
// DESCRIPTION:
// Main source file for the Align class. The Align class provides
// services related to aligning (synonymous with registering) a suspect
// image with a reference image. The suspect requires some combination
// of translation, scaling, and rotation to achieve this.
// This version incorporates the Version 1.0 Alignment core algorithms
// from Geoff Rhoads, 2/17/96.
// Copyright (C) Digimarc Corporation, 1996, all rights reserved.
//*****
#include <math.h>

#include <memory.h>
#include <stdarg.h>
#include "align.h"
#include "fft.h"

//*****
// Constructor for Align objects.
//*****
Align::Align()
{
    m_alignStatus.x_scale = (float) 0.0;
    m_alignStatus.y_scale = (float) 0.0;
    m_alignStatus.x_trans = (float) 0.0;
    m_alignStatus.y_trans = (float) 0.0;
    m_alignStatus.rotation = (float) 0.0;
    m_alignStatus.refinement = (float) 0.0;
}

//*****
// CORE ALGORITHMS FOLLOW
// The remainder of this file is devoted to the Align (i.e., register)
// core algorithms from Geoff Rhoads, modified slightly to comply with
// C++ and/or Windows programming standards.
//*****
#include <stdio.h>
#include <stdlib.h>

#define START_RADIUS 0.10 /* ratio of nyquist at which log scale vectors are started */
#define PICK_RADIUS 7 /* radius of samples to ignore around previously found candidates */
#define START_RADIUS_ID 0.07 /* ratio of nyquist at which log scale vectors are started */
#define MAX_CANDIDATES 20 /* this number can be set to 10 or even 50 when we start pushing things??? */
#define PI 3.141592653589
#define WINDOW_ORIGINALS 1
#define WINDOW_LOGPOLAR_LOG 1
#define MAX_LINEAR_DIMENSION 4096
#define MAX_SMALL (float) 1e-10
#define REFINED_ROTATION_DIMENSION 512
#define REFINED_ROTATION_BITS 9
#define LOG_MOV_AVG 27
#define LOG_SMOOTH 3
#define NOMINAL_DOWNSAMPLE_DIM 256
#define SUPER_DOWNSAMPLE_DIM 128
#define SIGNATURE_BLOCK_DIMENSION 128
#define MELLIN_DIMENSION 128

int lp_sampling = 128; /* total number of log-scale samples, should be plenty */
int lp_bits = 7; /* bit value of above line */
double scale_increment;

float wr[MAX_LINEAR_DIMENSION], wi[MAX_LINEAR_DIMENSION];

extern int reallft2d_in_place(float *ar, int nbits, int inv, float *wr, float *wi);
extern void fft(float *ar, float *ai, int nbits, int inv, float *wr, float *wi, int neww);
extern int load_bump_array(
    float *bump,
    unsigned char *data,
    long xdim,
    long ydim,
    long bump_size,
    long jump_x,
    long jump_y,
    long overfill
);

```

```

scale_increment=pow( 1.0/(double)START_RADIUS, 1.0/(double)lp_sampling);
for(i=0;i<lp_sampling;i++){
    radius[i] = (START_RADIUS*(double)dim2) * pow(scale_increment,(double)i);
}

pout = out;
for(theta=0.0;j=0;j<lp_sampling; j++){theta += (PI/lp_sampling)}{
    dx = cos(theta);
    dy = sin(theta);
    pradius = radius;
    pout = out[j];
    for(i=0;i<lp_sampling;i++){
        x = (double)dim2 * pradius * dx;
        y = (pradius**2) * dy;
        xx = (int)x;
        yy = (int)y;
        frack = x - (double)xx;
        frack = y - (double)yy;
        pin = sin(yy*dim + xx);
        *pout = (float) ( (1.0-frack) * (1.0-frack)*(double)*(pin**2) );
        *pout += (float) ( frack*(1.0-frack) * (double)*pin );
        pin = (dim-1);
        *pout += (float) ( (1.0-frack)*frack*(double)*(pin**2) );
        *pout += (float) ( frack*frack * (double)*pin );
        *pout += lp_sampling;
    }
}

/* now filter it along the scale axis */
/* this generally increases the peak to noise ratio in finding the proper scale rotation */
for(i=0;i<lp_sampling;i++){
    pout = ftemp;
    for(j=0;j<lp_sampling;j++){
        *pout = (float)0.0;
        for(k=(LOG_MOV_AVG/2);k<=(LOG_MOV_AVG/2);k++){
            j=j-k;
            if(j)<0;j=0;
            else if(j)>= lp_sampling;j=lp_sampling-1;
            *pout += out(i+j)*lp_sampling;
        }
        *pout += (float)LOG_MOV_AVG;
    }
    pin = ftemp;
    pout = out[i];
    for(j=0;j<lp_sampling;j++){
        *pout = (float)0.0;
        for(k=(LOG_SMOOTH/2);k<=(LOG_SMOOTH/2);k++){
            j=j-k;
            if(j)<0;j=0;
            else if(j)>= lp_sampling;j=lp_sampling-1;
            *pout += out(i+j)*lp_sampling;
        }
        *pout += (float)LOG_SMOOTH;
    }
    memcpy(out(i),ftemp,lp_sampling*sizeof(float));
}

return(i);
}

float get_median_float(float *median){
    float *x_offset, float *y_offset;
    int j, jtemp, k, ktemp;
    ymedian[0]=ymedian[1]=ymedian[2]=(float)0.0;
    py = ymedian;
    for(j=1;j<2;j++){
        jtemp = high_y+j;
        if(jtemp < 0)jtemp=ydim-1;
        else if(jtemp==ydim)jtemp=0;
        px = xmedian;
        for(k=1;k<2;k++){
            ktemp = high_x+k;
            if(ktemp < 0)ktemp=xdim-1;
            else if(ktemp==xdim)ktemp=0;
            *py += array[jtemp*xdim+ktemp];
            *px += array[jtemp*xdim+ktemp];
            py++;
        }
        /* now find median values */
        ratio = get_median_float(ymedian);
        *y_offset = (float)high_y * ratio;
        ratio = get_median_float(xmedian);
        *x_offset = (float)high_x * ratio;
        value = (xmedian[0]*xmedian[1]*xmedian[2])/(float)9.0;
        return(value);
    }

    /* this is the fft window profile for mitigating edge effects; change to other windows if
    their better */
    /* or... maybe certain windows are better for certain tasks, e.g., log polar vs. straight
    correlation */
    int load_windowing_function(int dim,float *window){
        int i;
        double step,x,y;

        step = 2.0*PI / (double)(dim+1);
        for(i=0,x=step;i<dim;i++,x+=step){
            y = (1.0 - cos(x))/2.0;
            window[i] = (float)sqrt(y);
        }
        return(i);
    }

    int window_1d_vector(
        float *array,
        int data_length,
        int full_length
    ){
        int i;
        float *parray,*pwindow;

        float *window_function = new float(data_length);
        load_windowing_function(data_length,window_function);
        parray = array;
        pwindow = window_function;
        for(i=0;i<data_length;i++){parray++} * * (pwindow++);
        if(full_length != data_length){
            for(i=0;i<(full_length - data_length);i++){parray++} * (float)0.0;
        }
        delete [] window_function;
        return(i);
    }

    /* this module specifically designed for the rough thumbnail registration
    on the pixels, but now think this is overkill because of the later refinement
    anyway, who knows */
    int rotate_scale_translate_image(
        float *out,
        int outdim,
        float *in,
        int inxdim,
        int inydim,
        int orig_xdim,
        int orig_ydim,
        int downsample,
        float rotation,
        float scale
    ){
        int i,j,xx,yy;
        float a_const,b_const,x,y,dx,dy,*pout;
        float middle_in_x, middle_in_y,middle_out;

        /* make sure to place the center of the original array at the center of
        the output array; this helps later translation bookkeeping */
        middle_in_x = (float)(orig_xdim - downsample)/(float)downsample/(float)2.0;
        middle_in_y = (float)(orig_ydim - downsample)/(float)downsample/(float)2.0;
        middle_out = (float)(outdim-1)/(float)2.0;
        rotation = rotation; //who can keep track of CW and CCW anyway???
        a_const = (float)cos((double)rotation*PI/180.0)*scale;
        b_const = (float)sin((double)rotation*PI/180.0)*scale;
        dx = a_const;
        dy = b_const;
        pout = out;
        for(i=0;i<outdim;i++){
            x = middle_in_x - a_const*middle_out + b_const*(middle_out-(float)i) + (float)0.5;
            y = middle_in_y - b_const*middle_out - a_const*(middle_out-(float)i) + (float)0.5;

```

```

for(j=0;j<outdim;j++){
    if(x<(float)0.5){
        x=(float)(inxdim-1)*(float)0.5/(float)0.5;
    }
    else{
        xx = (int)x;
        yy = (int)y;
        *pout++ = inly*outdim+xx;
    }
    x=dx;
    y=dy;
}
return(1);
}

/* step through the found candidates, finding inter-sample values for the peak location */
for(i=0;i<number_candidates;i++){
    ymedian[0]=ymedian[1]=ymedian[2]=(float)0.0;
    xmedian[0]=xmedian[1]=xmedian[2]=(float)0.0;
    py = ymedian;
    for(j=-1;j<2;j++){
        if(jtemp < 0){jtemp=dim-1;
        else if(jtemp==dim)jtemp=0;
        px = xmedian;
        for(k=-1;k<2;k++){
            ktemp = x_off[i]*k;
            if(ktemp < 0)ktemp=dim-1;
            else if(ktemp==dim)ktemp=0;
            *py++ = reall[jtemp*dim+ktemp];
            *px++ = reall[jtemp*dim+ktemp];
        }
        py++;
    }
}

/* now find median values */
ratio = get_median_float(ymedian);
y_offest[i] = (float)dim2 - ((float)y_off[i] + ratio);
ratio = get_median_float(xmedian);
x_offest[i] = (float)dim2 - ((float)x_off[i] + ratio);
value[i] = reall[x_off[i] + dim*y_off[i]];
}
return(1);
}

/* simple sub-routine for direct_registration
int get_working_dimension(
    int alignment_mode,
    int xdim1,
    int ydim1,
    int xdim2,
    int ydim2,
    int *downsample
){
    int highest=xdim1,go=1,fftdim;

    if(ydim1>highest)highest = ydim1;
    if(xdim2>highest)highest = xdim2;
    if(ydim2>highest)highest = ydim2;

    switch(alignment_mode){
        case 0: // no downsampling
            *downsample = 1;
            fftdim = 1;
            while(go){
                if(highest > fftdim){
                    fftdim*=2;
                }
                else go = 0;
            }
            break;
        case 1: // nominal downsampling
            *downsample = ((highest-1)/NOMINAL_DOWNSAMPLE_DIM)+1;
            fftdim = NOMINAL_DOWNSAMPLE_DIM;
            break;
        case 2: // super downsampling
            *downsample = ((highest-1)/SUPER_DOWNSAMPLE_DIM)+1;
            fftdim = SUPER_DOWNSAMPLE_DIM;
            break;
    }
    return(fftdim);
}

/* another sub-routine for direct_registration
int copy_downsample_window(
    unsigned char *in,
    int xdim,
    int ydim,
    float *out,
    int outdim,
    int *downsample
){
    unsigned char *pin;
    int i,j;
}

```



```

)
*(pimaginary1++) = cross*dott;
}

fft(reall.imaginary,bits,1,wr,w1,1);
/* search for highest value, then median find the center */
highest = (float)1e20;
preall = reall;
for(i=0; i<xdim; i++){
    if( preall > highest){
        highest = preall;
        highest_i = i;
    }
    preall++;
}

if(highest_i == 0){
    median[0]=reall[highest_i-1];
    median[1]=reall[0];
    median[2]=reall[1];
}
else if(highest_i == (dlim-1)){
    median[0]=reall[dlim-2];
    median[1]=reall[dlim-1];
    median[2]=reall[0];
}
else {
    median[0]=reall[highest_i-1];
    median[1]=reall[highest_i+1];
    median[2]=reall[highest_i];
}
ratio = get_median_float(median);
offset = (float)highest_i * ratio;
if( *offset > (float)dlim/2.0 ) *offset -= (float)dlim;
return(1);
}

int refine_axis(
    unsigned char *template,
    int template_xdim,
    int template_ydim,
    unsigned char *suspect,
    int suspect_xdim,
    int suspect_ydim,
    float *x,
    float *y,
    int which
){
    int i,j,highest,fftdim,bits,xx,yy,xdim,ydim;
    float x0,x1,x2,y0,y1,y2,*psuspect_integral,*ptemplate_integral;
    float scan_x,scan_y,jump_x,jump_y,current_x,current_y;
    float scale,translation,distance,ydistance,suspect_dc,template_dc,frac;
    double scale_increment_id;

    /* first convert the y axis version to the x axis version */
    x0 = x[0]; y0 = y[0];
    if(which){
        x1 = x[2]; y1 = y[2];
        x2 = x[1]; y2 = y[1];
        xdim = suspect_ydim;
        ydim = suspect_xdim;
    }
    else {
        x1 = x[1]; y1 = y[1];
        x2 = x[2]; y2 = y[2];
        xdim = suspect_xdim;
        ydim = suspect_ydim;
    }

    /* determine the next highest power of two above higher of the two suspect axes */
    if(suspect_xdim > suspect_ydim) highest = suspect_xdim;
    else highest = suspect_ydim;
    bits = 1 + (int)( log( (double)highest - 0.5 ) / log(2.0) );
    fftdim = (int)pow(2.0,(double)bits + 0.00000001);

    float *template_integral = new float[fftdim];
    float *suspect_integral = new float[fftdim];
    float *template_imaginary = new float[fftdim];
    float *suspect_imaginary = new float[fftdim];
    float *template_copy = new float[fftdim];
    float *suspect_copy = new float[fftdim];

    /* load suspect integral waveform */
    psuspect_integral = suspect_integral;
    for(j=0; j<fftdim; j++){ *psuspect_integral++ = (float)0.0;
        if(!which){
            psuspect = suspect;

```

```

convert_to_magnitude_id(inplace(suspect_integral, suspect_integral_imaginary, fftdim);
// next routine places output into integral_imaginary array
scale_increment_id = log_id_remop(suspect_integral, suspect_integral_imaginary, fftdim);
scale_increment_id = log_id_remop(template_integral, template_integral_imaginary, fftdim);
// copy output back into fundamental array and zero out imaginary
memcpy(suspect_integral, suspect_integral_imaginary, sizeof(float)*fttdim);
memcpy(template_integral, template_integral_imaginary, sizeof(float)*fttdim);
memset(suspect_integral_imaginary, 0, sizeof(float)*fttdim);
memset(template_integral_imaginary, 0, sizeof(float)*fttdim);
// now do the id fourier mullin trot
window_id_vector(template_integral, fttdim, fftdim);
window_id_vector(suspect_integral, fttdim, fftdim);
fft(suspect_integral, suspect_integral_imaginary, bits, 0, wr, wi, 1);
fft(template_integral, template_integral_imaginary, bits, 0, wr, wi, 1);
// gm_fld to find any small scaling difference between the two
gm_fld(suspect_integral, suspect_integral_imaginary, template_integral,
template_integral_imaginary, fftdim, bits, kscale);
// scale = (float)0.6; // slight damping factor
scale = (float)pow(scale_increment_id, (double)scale);
// update the x's and y's
xdistance = (xi-x0);
ydistance = ((float)1.0 - scale);
ydistance = (yi-y0);
x[3] += xdistance; y[3] += ydistance;
x[4] += xdistance/(float)2.0; y[4] += ydistance/(float)2.0;
if (which) {
    x[2] += xdistance; y[2] += ydistance;
    x1 = x[2]; y1 = y[2];
} else {
    x[1] += xdistance; y[1] += ydistance;
    x1 = x[1]; y1 = y[1];
}
// now with the new scale information, perform a gm_f on the original and its rescaled
counterpart.
template_integral = template_integral;
scale = (float)1.0 / scale;
float lllast;
for (i=0; current_x=(float)0.0; i<xdim; i++, current_x+=scale) {
    xx = (int)current_x;
    if (ix >= xdim-1) { ptemplate_integral++ } = lllast;
    else {
        frac = current_x - (float)xx;
        *ptemplate_integral = ((float)1.0-frac) * template_integral_copy[xx];
        *ptemplate_integral++ += frac * template_integral_copy[xx+1];
    }
    lllast = *(ptemplate_integral-1);
}
// window the new scaled array; other one should be copy of windowed original
memcpy(suspect_integral, suspect_integral_copy, sizeof(float)*fttdim);
window_id_vector(template_integral, xdim, fftdim);
window_id_vector(suspect_integral, xdim, fftdim);
memset(template_integral_imaginary, 0, sizeof(float)*fttdim);
memset(suspect_integral_imaginary, 0, sizeof(float)*fttdim);
fft(suspect_integral, suspect_integral_imaginary, bits, 0, wr, wi, 1);
fft(template_integral, template_integral_imaginary, bits, 0, wr, wi, 1);
// now find the translation
gm_fld(suspect_integral, suspect_integral_imaginary, template_integral,
template_integral_imaginary, fftdim, bits, ktranslation);
// adjust x and y accordingly
translation = (float)0.5;
origins???? very kludge
scan_x = translation;
scan_y = translation;
x[0] += scan_x; y[0] += scan_y;
x[1] += scan_x; y[1] += scan_y;
x[2] += scan_x; y[2] += scan_y;
x[3] += scan_x; y[3] += scan_y;
x[4] += scan_x; y[4] += scan_y;
delete {} template_integral;
delete {} suspect_integral;
delete {} template_integral_imaginary;
delete {} suspect_integral_imaginary;
delete {} template_integral_copy;
delete {} suspect_integral_copy;
return(0);
}
float refined_rotation(

```

```

float *x,
float *y,
unsigned char *suspect,
int suspect_xdim,
int suspect_ydim,
unsigned char *template,
int template_xdim,
int template_ydim
) {
    int i, xx, yy, count, template_count, suspect;
    float line_integral[REFINED_ROTATION_DIMENSION], *pli, *pli_template;
    float line_integral[REFINED_ROTATION_DIMENSION];
    float line_integral_imaginary[REFINED_ROTATION_DIMENSION];
    float line_integral_imaginary[REFINED_ROTATION_DIMENSION];
    float angle, x, suspect, y, suspect_x1, suspect_y1, suspect_dx, suspect_dy, suspect;
    float x_template, y_template, x1_template, y1_template, dx_template, dy_template;
    float top_x, suspect=(float)(suspect_xdim-1), top_y, suspect=(float)(suspect_ydim-1);
    float top_x_template=(float)(template_xdim-1), top_y_template=(float)(template_ydim-1);
    float a_const, b_const, tweak, dc_suspect, dc_template;
    float new_x, new_y, yaxis_y, yaxis_x, xaxis_x, xaxis_y;
    yaxis_x = (x[2]-x[0])/(float)(suspect_ydim-1); // this gives the unit vector in terms of
the suspect array.
    yaxis_y = (y[2]-y[0])/(float)(suspect_ydim-1);
    xaxis_x = (x[1]-x[0])/(float)(suspect_xdim-1);
    xaxis_y = (y[1]-y[0])/(float)(suspect_xdim-1);
    // create line integral sweep around suspect's and template's center point.
    pli = line_integral;
    pli_template = line_integral;
    dc_suspect = dc_template=(float)0.0;
    for (i=0; i<REFINED_ROTATION_DIMENSION; i++) {
        angle = (float)i * (float)PI / (float)REFINED_ROTATION_DIMENSION;
        x_suspect = x1_suspect = (float)0.5 + top_x_suspect/(float)2.0;
        y_suspect = y1_suspect = (float)0.5 + top_y_suspect/(float)2.0;
        dx_suspect = (float)sin((double)angle);
        dy_suspect = (float)cos((double)angle);
        x_suspect+=dx_suspect; x1_suspect+=dx_suspect;
        y_suspect+=dy_suspect; y1_suspect+=dy_suspect;
        x_template = x1_template = (float)0.5+x[4];
        y_template = y1_template = (float)0.5+y[4];
        dx_template = (xaxis_x*dx_suspect+yaxis_x*dy_suspect);
        dy_template = (xaxis_y*dx_suspect+yaxis_y*dy_suspect);
        x_template+=dx_template; x1_template+=dx_template;
        y_template+=dy_template; y1_template+=dy_template;
        *pli = (float)0.0;
        count_template=0; count_suspect=0;
        while (x_suspect>0.0 && x_suspect<top_x_suspect && y_suspect>0.0 &&
y_suspect<top_y_suspect) {
            xx = (int)x_suspect;
            yy = (int)y_suspect;
            *pli += suspect[yy*suspect_xdim+xx];
            xx = (int)x1_suspect;
            yy = (int)y1_suspect;
            *pli += suspect[yy*suspect_xdim+xx];
            x_suspect+=dx_suspect; x1_suspect+=dx_suspect;
            y_suspect+=dy_suspect; y1_suspect+=dy_suspect;
            count_suspect++;
        }
        if (y_template>0.0 && y_template<top_y_template && x_template>0.0 && x_template<top_x_template) {
            xx = (int)x_template;
            yy = (int)y_template;
            *pli_template += template[yy*template_xdim+xx];
            xx = (int)x1_template;
            yy = (int)y1_template;
            *pli_template += template[yy*template_xdim+xx];
            x_template+=dx_template; x1_template+=dx_template;
            y_template+=dy_template; y1_template+=dy_template;
            count_template++;
        }
        *pli /= (float)count_suspect;
        *pli_template /= (float)count_template;
        dc_suspect += *pli++;
        dc_template += *pli_template++;
    }
    // now one-d fft them and one d gm_f
memset(line_integral_imaginary, 0, sizeof(float)*REFINED_ROTATION_DIMENSION);

```

```

memset(line_integral_template_imaginary,0,sizeof(float))*REFINED_ROTATION_DIMENSION);
pli = line_integral;
pli_template = line_integral_template;
dc_suspect /= (float)REFINED_ROTATION_DIMENSION;
dc_template /= (float)REFINED_ROTATION_DIMENSION;
for(i=0;i<REFINED_ROTATION_DIMENSION;i++){
    *pli++ = dc_suspect;
    *pli_template++ = dc_template;
}
fft(line_integral,line_integral_imaginary,REFINED_ROTATION_BITS,0,wr,wi,1);
fft(line_integral_template,line_integral_template_imaginary,REFINED_ROTATION_BITS,0,wr,wi,1);

gmf_id(line_integral,line_integral_imaginary,line_integral_template,line_integral_template_imaginary,
REFINED_ROTATION_DIMENSION,REFINED_ROTATION_BITS,&tweak);
tweak *= (float)0.5; // slight damping factor

tweak *= -(float)180.0/(float)REFINED_ROTATION_DIMENSION);
/* update xy0 thru xy3 */
a_const = (float)cos( (double)tweak * PI /180.0 );
b_const = (float)sin( (double)tweak * PI /180.0 );
new_x = a_const*(x[4]-x[0]) - b_const*(y[4]-y[0]);
new_y = b_const*(x[4]-x[0]) + a_const*(y[4]-y[0]);
x[0] = x[4] - new_x;
y[0] = y[4] - new_y;
new_x = a_const*(x[4]-x[1]) - b_const*(y[4]-y[1]);
new_y = b_const*(x[4]-x[1]) + a_const*(y[4]-y[1]);
x[1] = x[4] - new_x;
y[1] = y[4] - new_y;
new_x = a_const*(x[4]-x[2]) - b_const*(y[4]-y[2]);
new_y = b_const*(x[4]-x[2]) + a_const*(y[4]-y[2]);
x[2] = x[4] - new_x;
y[2] = y[4] - new_y;
new_x = a_const*(x[4]-x[3]) - b_const*(y[4]-y[3]);
new_y = b_const*(x[4]-x[3]) + a_const*(y[4]-y[3]);
x[3] = x[4] - new_x;
y[3] = y[4] - new_y;
return(tweak);
}

int Align::fine_tune_x_y(unsigned char *template,
int template_xdim,
int template_ydim,
unsigned char *suspect,
int suspect_xdim,
int suspect_ydim,
float *x,
float *y,
float *rotation)
{
//int foo=1;
float refinement;

//while(foo){
// find xscale, ytrans optimal pair */
refine_axis(template,template_xdim,template_ydim,suspect,suspect_xdim,
suspect_ydim,x,y,0);
/* find yscale, ytrans optimal pair */
refine_axis(template,template_xdim,template_ydim,suspect,suspect_xdim,
suspect_ydim,x,y,1);
/* fine tune rotation */
refinement = refined_rotation(x,y,suspect,suspect_xdim,suspect_ydim,template,
template_xdim,template_ydim);
// NOTES: SOME CONFUSION ABOUT YDIM;
*rotation += refinement;
//}

m_alignStatus.refinement = refinement;
return(1);
}

/* subroutine for direct registration */
int get_corners_and_center(
float *x,
float *y,
float rotation,
float scale,
float x_trans,
float y_trans,
int xdim,
int ydim,
int ftdim,
int downsample)
{
float a_const,b_const;
/* the center of the suspect array should translate to... y??? */
//((float)downsample - 1)/2.0 - x_trans*downsample, same on y??? */
/* note that the origin of the downsampled arrays actually is
positioned at (downsample-1)/2, (downsample-1)/2 in the coordinates of the
original arrays */
x_trans *= (float)downsample;
y_trans *= (float)downsample;
x[4] = (float)(ftdim*downsample - 1)/(float)2.0 + x_trans;
y[4] = (float)(ftdim*downsample - 1)/(float)2.0 + y_trans;
a_const = (float)cos((double)rotation*PI/180.0)/scale;
b_const = (float)sin((double)rotation*PI/180.0)/scale;
x[0] = x[4] - (a_const*(float)(xdim-1) - b_const*(float)(ydim-1))/(float)2.0;
y[0] = y[4] - (b_const*(float)(xdim-1) + a_const*(float)(ydim-1))/(float)2.0;
x[1] = x[4] + (a_const*(float)(xdim-1) + b_const*(float)(ydim-1))/(float)2.0;
y[1] = y[4] + (b_const*(float)(xdim-1) - a_const*(float)(ydim-1))/(float)2.0;
x[2] = x[4] - (a_const*(float)(xdim-1) + b_const*(float)(ydim-1))/(float)2.0;
y[2] = y[4] - (b_const*(float)(xdim-1) - a_const*(float)(ydim-1))/(float)2.0;
x[3] = x[4] + (a_const*(float)(xdim-1) - b_const*(float)(ydim-1))/(float)2.0;
y[3] = y[4] + (b_const*(float)(xdim-1) + a_const*(float)(ydim-1))/(float)2.0;
return(1);
}

int final_image(
unsigned char *out,
int outxdim,
int outydim,
unsigned char *in,
int inxdim,
int inydim,
float *x,
float *y,
int num_channels,
int option)
{
unsigned char *pout;
int i,j,xx,yy;
float ii,current_x,current_y,fracy,ftmp,ftmp1,ftmp2,ftmp3,ftmp4;
float xxaxis_x,axis_y,xxaxis_y,axis_dist,axis_dist;
float x_start,y_start,scan_x,scan_y,jump_x,jump_y;
unsigned char *pin;
if(option==1){ // clear ttemplate array
pout=out;
for(i=0;i<(num_channels*outxdim*outydim);i++){*(pout++)=(unsigned char)0;
}
}
xxaxis_x = (x[2]-x[0])/(float)(inydim-1); // this gives the unit vector in terms of the
suspect array
xxaxis_y = (y[2]-y[0])/(float)(inydim-1);
axis_dist = (float)sqrt((double)(xxaxis_x*xxaxis_x+xxaxis_y*xxaxis_y));
xxaxis_x = (x[1]-x[0])/(float)(inxdim-1);
xxaxis_y = (y[1]-y[0])/(float)(inxdim-1);
axis_dist = (float)sqrt((double)(xxaxis_x*xxaxis_x+xxaxis_y*xxaxis_y));
/* starts is origin dotted with axes */
x_start = (-x[0] * xxaxis_x - y[0] * xxaxis_y)/axis_dist/xxaxis_dist;
y_start = (-x[0] * xxaxis_y + y[0] * xxaxis_x)/axis_dist/xxaxis_dist;
scan_x = xaxis_x/axis_dist/xxaxis_dist;
scan_y = yaxis_y/axis_dist/xxaxis_dist;
jump_x = xaxis_x/axis_dist/xxaxis_dist;
jump_y = yaxis_y/axis_dist/xxaxis_dist;
pout = out;
for(i=0;i<(outydim;i++){
    ii = (float)i;
    current_x = x_start + ii * jump_x;
    current_y = y_start + ii * jump_y;
    if(num_channels==1){
        for(j=0;j<(outxdim;j++){
            if(current_x<(float)0.0 || current_x>(float)(inxdim-1) || current_y<(float)0.0
|| current_y>(float)(inydim-1)){
                if(option==0)pout++; // this option preserves the rest of template
                else *(pout++) = (unsigned char)0;
            }
            else {
                xx = (int)current_x;
                yy = (int)current_y;
                fracy = current_x - (float)xx;
                fracy = current_y - (float)yy;
            }
        }
    }
}
}

```

```

double start = sqrt(32.5);
for(i=0; i<n; i++) {
    radius[i] = start * pow(increment, (double)i);
    // pre-filter fourier mag data;
    // first add 90 degree separated points for 2root2 improvement
    for(j=0; j<64; j++) { // output into left half of original array
        in[63-i+128+j] = in[(1+i)*128-64+j];
    }
}

float local_average, *p1, *p2, *p3;
for(i=0; i<64; i++) {
    pout = &in[64+129+i], // output into right half of original array
    if(i==0) p1 = in;
    else p1 = &in[(i-1)*128];
    p2 = &in[(i-1)*128];
    if(i==63) p3 = &in[63+128];
    else p3 = &in[(i+1)*128];
    // first element
    local_average = (*p1 + *p2 + *p3) / 3.0;
    if(*p2 > (float)100.0 * local_average) {
        *pout = (float)100.0 * local_average;
    } else {
        *pout = *p2 / local_average;
    }
    // last element
    local_average = (*p1 + *p2 + *p3) / 3.0;
    if(*p2 < (float)100.0 * local_average) {
        *pout = (float)100.0 * local_average;
    } else {
        *pout = *p2 / local_average;
    }
}

// copy horizontal row into vertical column for interp porpoises
for(i=1; i<64; i++) in[64+i] = in[64+i*128];

pout = out;
for(theta=0.0; theta<n; theta += PI/((double)n)/2.0) {
    dx = cos(theta);
    dy = sin(theta);
    pradius = radius;
    pout = &out[j];
    for(i=0; i<n; i++) {
        x = (double)dim2 * pradius * dx;
        y = (pradius**2) * dy;
        xy = (int)x;
        fracy = (int)y;
        pin = &in[y*dim - xx];
        *pout++ = (float) ((1.0-fracy)*(1.0-fracy)* (double)(pin**2));
        *pout++ = (float) (fracy*(1.0-fracy)* (double)pin);
        *pout++ = (float) ((1.0-fracy)*fracy* (double)(pin**2));
        *pout++ = (float) (fracy*fracy* (double)pin);
    }
}

return(1);
}

load_grid_family()
{
    static int done = 0;
    // don't change this without checking its effects on the later grid finding routines
    // such as resolve_orientation

```





```

int xdim,
int ydim,
int zdim,
int bump_size,
int n,
int original_xdim,
float rotation,
float scale,
float *out
){
    int n2 = n/2;
    float outcenter = (float)(n-1) / (float)2.0;
    float incenter = (float)(xdim-1) / (float)2.0;
    float incenter = (float)(ydim-1) / (float)2.0;

    // create buffer for input data
    float *buffer = new float[xdim*ydim];

    // load buffer array with bump data
    unsigned char *pdata = data;
    float *pbuffer;
    int i;
    for(i=0; i<ydim; i++){
        pbuffer = &buffer(i*n);
        load_bump_array(
            pbuffer, // floating point bump array to be filled (output)
            pdata, // input pixel data
            xdim, // number of bumps in this row (not pixels)
            zdim, // number of channels
            bump_size, // pixels per bump
            original_xdim - xdim*bump_size, // number of raw pixels between (xdim*bump_size) and entire
            image array x dimension
            0 // do not overfill the bump buffer
        );
        pdata += (zdim*original_xdim*bump_size);
    }

    // now rotate and scale the input image inside buffer, into the output image
    // use xdim/2 and ydim/2 as the center of rotation for the input image
    // use n/2 and n/2 as the center of rotation for the output array
    scale = (float)1.0 / scale;
    rotation = -rotation;
    float costheta = scale * (float)cos( (double) rotation * PI / 180.0 );
    float sintheta = scale * (float)sin( (double) rotation * PI / 180.0 );
    float ii,jj,fracx,fracy,*pout = out,*pin,x,y;
    int xx,yy,j;
    for(i=0; i<n; i++){
        ii = (float)i - outcenter;
        for(j=0; j<n; j++){
            jj = (float)j - outcenter;
            x = ii * costheta + jj * sintheta;
            y = ii * costheta - jj * sintheta;
            x+=incenter;
            y+=incenter;
            xx = (int)x;
            yy = (int)y;
            if(xx < 0){
                xx = 0;
                fracx = (float)0.0;
            }
            else if(xx >= xdim-1){
                xx = xdim-1;
                fracx = (float)1.0;
            }
            else fracx = x - (float)xx;
            if(yy < 0){
                yy = 0;
                fracy = (float)0.0;
            }
            else if(yy >= ydim-1){
                yy = ydim-1;
                fracy = (float)1.0;
            }
            else fracy = y - (float)yy;

            pin = &buffer(yy*n + xx);
            *pout = ( (float)1.0-fracy)*((float)1.0-fracy)* * (pin++) );
            *pout += (fracx*((float)1.0-fracy)* *pin);
            pin += (n-1);
            *pout += ( (float)1.0-fracy)*fracy* * (pin++) );
            *(pout++) += (fracx*fracy* *pin);
        }
    }

    delete [] buffer;

    return(1);
}

```

// this is a specialized function simply meant to find out which of 4  
 possible orientations is the true orientation of the subliminal grid;  
 the function takes the 2D FFT of the image, and performs a 2D  
 discrete Fourier transform, combined with our "folding" of frequencies,  
 gives this ambiguity in the first place

```

int resolve_orientation(
    unsigned char *data,
    int xdim,
    int ydim,
    int zdim,
    int bump_size,
    int n, // power of 2 used in inverse fft's
    int original_xdim,
    float rotation,
    float *scale
){
    int mult = 1;
    if(*scale > (float)1.25){ // up n to the next higher power of two
        n*=2;
        mult = 2;
    }

    float *buffer = new float[n*(n+2)];
    int n2 = n/2+1;

    rotate_scale_image(
        data,
        xdim,
        ydim,
        zdim,
        bump_size,
        n,
        original_xdim,
        *rotation,
        *scale,
        buffer
    );
    // fft the thing
    int bits = (int) (log( (double)(n+1) ) / log( 2.0 ) ); // fftdim should always be power
    of 2
    realfft2d_in_place(buffer, bits, 0, vr, wi); // ultimately, direct calculation may be faster
    assuming frequency points < bits*bits

```

```

    // save the original phase values
    float *real = new float[grid_freq_total];
    float *imag = new float[grid_freq_total];
    for(i=0; i<grid_freq_total; i++){
        real[i] = buffer[n2 + mult*grid_x[i] + 2*n*mult*grid_y[i]];
        imag[i] = -buffer[n + n2 + mult*grid_x[i] + 2*n*mult*grid_y[i]];
    }

```

// now step through the four possible orientations, finding the best fit  
 // the current incarnation of this routine is intimately tied to  
 // the function load\_grid\_family

```

    float highest_high = (float)-1e20; grid_real, grid_imag;
    int highi, tmp;
    float value[4], x_offset[4], y_offset[4];
    for(i=0; i<4; i++){
        // zero out buffer
        memset(buffer, 0, sizeof(float)*n*(n+2));
        // multiply this orientation by saved phases
        for(j=0; j<grid_freq_total; j++){
            if(i==0){
                grid_real = (float)cos( (double)grid_phase[j]);
                grid_imag = (float)sin( (double)grid_phase[j]);
            }
            else if(i==1){
                tmp = (j-grid_freq_total/2)%grid_freq_total;
                grid_real = (float)cos( (double)grid_phase[tmp]);
                if(tmp >= grid_freq_total/2) grid_imag = (float)sin( (double)grid_phase[tmp]);
                else grid_imag = -(float)sin( (double)grid_phase[tmp]);
            }
            else if(i==2){
                grid_real = (float)cos( (double)grid_phase[j]);
                grid_imag = -(float)sin( (double)grid_phase[j]);
            }
            else if(i==3){
                tmp = (j-grid_freq_total/2)%grid_freq_total;
                grid_real = (float)cos( (double)grid_phase[tmp]);
                if(tmp >= grid_freq_total/2) grid_imag = -(float)sin( (double)grid_phase[tmp]);
                else grid_imag = (float)sin( (double)grid_phase[tmp]);
            }
        }
        buffer[n2 + mult*grid_x[j] + 2*n*mult*grid_y[j]] = real[j] * grid_real -
        imag[j] * grid_imag;
        buffer[n + n2 + mult*grid_x[j] + 2*n*mult*grid_y[j]] = real[j] * grid_imag +
        imag[j] * grid_real;
    }

```

```

    }
    realfft2d_in_place(buffer.bits,1,wr,w1); // ultimately, direct calculation may be faster
    assuming frequency points < bits*bits

    // find highest point
    highest = (float)-1e20;
    float *pbuffer = buffer;
    int high_x, high_y;
    for(j=0;j<(n-n);j++){
        if( *pbuffer > highest ){
            highest = *pbuffer;
            high_y = j/n;
            high_x = j - high_y;
        }
        pbuffer++;
    }

    // load its median inter-sample value
    value[i] = get_2D_median(buffer,n,n,high_x,high_y,wx_offset[i],wy_offset[i]);

    // then, find the highest of the four
    if(highest > high){
        high1 = i;
        high = highest;
    }

    // update rotation
    *rotation += (float)high1 * (float)90.0;
    delete () real;
    delete () imag;
    return(i);
}

/*
This function performs two basic services: first, it simply attempts
to determine if a public subliminal grid exists or not;
if one does exist, then the second basic service is to determine the
rough scale and rotation state of that grid.

The mode_flag variable provides options for how fast v. thorough the algorithms
are.
*/
int hunt_for_grid(
    unsigned char *data, // input image, unknown signature status
    int xdim, // its full pixel dimension in x
    int ydim, // ditto in y
    int zdim, // number of channels
    int probable_bump_size, // this is a tricky one to start: to best function,
    // we will need to specify or "recommend" some standard bumps-per-inch
    // and first look for the signatures in that region
    int total_blocks, // how hard do we look
    int *present,
    float *scale,
    float *rotation,
    float *mellin_mag_transform
){
    int xblocks,yblocks,i,j,xlength,ylength;
    unsigned char *pdata;

    // the checking takes the first N 128x128 bump regions, FFT's them,
    // converts them to magnitudes, adds them all, then does
    // the fourier \mellin check between the added versions and
    // the master public grid PM profile.
    // A Yea/No is generated based on the S/N found between a peak and the
    // background

    // find and use full integral blocks only, unless the data is shorter
    // than a full integral block
    int xbumpsizes = xdim/probable_bump_size;
    int ybumpsizes = ydim/probable_bump_size;
    xblocks = xbumpsizes / SIGNATURE_BLOCK_DIMENSION; // if 0, doesn't even cover one block but will
    still function
    yblocks = ybumpsizes / SIGNATURE_BLOCK_DIMENSION;

    // temporary
    total_blocks = xblocks * yblocks; // again, 0 will function

    // create the basic fourier magnitude array (SIGDIM*(SIGDIM/2+1)) or 128 by 65
    int n=SIGNATURE_BLOCK_DIMENSION;
    float *fourier_mag = new float(n*(n/2+1)); // only stores the magnitude
    float *buffer = new float(n*(n/2+1)); // give it a full array for processing inside 'add_block'
    int m = MELLIN_DIMENSION;

```

```

    float *mellin_mag = new float(m*(m/2+1));
    float f0 = (float)0.0;
    for(i=0;i<(n*(n/2+1));i++){fourier_mag[i]=f0;
    // f0 = 0.0;
    int count = 0;
    for(i=0;i<yblocks;i++){
        count++;
        pdata = &data((i*xdim+j)*n+probable_bump_size); // offset to this block
        if(xblocks == 0 || yblocks == 0){
            truncated = 1;
            if(xblocks==0)xlength = xbumpsizes;
            else xlength = n;
            if(yblocks==0)ylength = ybumpsizes;
            else ylength = n;
        }
        else {
            truncated = 0;
            xlength = n;
            ylength = n;
        }
        add_block_magnitude(
            pdata,
            fourier_mag,
            n,
            buffer,
            xlength,
            ylength,
            probable_bump_size,
            xdim, // pixel based jump pointer for moving down rows
            truncated
        );
        if(count >= total_blocks){xblocks=i*yblocks;}//this kicks it out
    }

    // temporary: ship this one back for display
    // use atemp.bmp as input alignment template file
    // memcpy(mellin_mag_transform,fourier_mag,sizeof(float)*n*(n/2+1));
    // return(i);

    // now fourier mellinize the magnitude profile
    log_polar_temp_publicfourier_mag,mellin_mag,n);
    // temporary display results code
    // use atemp128.bmp as input alignment template file
    // memcpy(mellin_mag_transform,mellin_mag,sizeof(float)*n*n);
    // return(i);

    // fourier transform the dog
    realfft2d_in_place(mellin_mag,7,0,wr,w1);

    load_grid_family(i); // will immediately return if already done
    // temporary display results code: this one has a corresponding return inside
    load_grid_family(i);
    // memcpy(mellin_mag_transform,subliminal_grid,sizeof(float)*128*128);
    // return(i);

    // now compare the patterns
    of 2
    int bits = (int) (log( double) (n+1) / log( 2.0 ) ); // fftdim should always be power
    int number_candidates = 20;
    float *rotation_buf = new float(number_candidates);
    float *scale_buf = new float(number_candidates);
    float *value = new float(number_candidates);
    gmf(mellin_mag,mellin_mag_transform,n,bits,number_candidates,rotation_buf,scale_buf,value,0);
    // return(i);

    // a first crack at deciding whether or not a signature/grid is present is possible
    // at this point: the ratio between value and value should be above some
    // threshold. If this is unreliable, then complete the alignment/read process.
    // read the control bits and their checksums, and see if the checksums are right;
    // this will obviously take a longer time to make a negative decision.

    delete () fourier_mag;
    delete () buffer;
    delete () mellin_mag;

    float detection_value = value[0] / value[19];
    float threshold_detect = (float)2.0; // where's our empirical data anyway, false-positive
    curves, true double entendre negatives, etc.
    if(detection_value > threshold_detect){ // we have a winna
        // if the suspect image has been rotated clockwise, rotation_buf will be positive
        // if the suspect image has been expanded, scale will come back negative
        rotation_buf[0] = (float)(90.0 / 128.0);
        double increment = pow( 2.0 , 0.025);

```

```

scale_buf[0] = (float)pow(increment, (double)scale_buf[0]);
if(xblocks == 0 || yblocks == 0){
    truncated = 1;
    if(xblocks==0)length = xbumpsiz;
    else length = n;
    if(yblocks==0)length = ybumpsiz;
    else length = n;
}
// resolve 90 degree ambiguity in rotation/orientation
resolve_orientation(data, xlength, ylength, xdim, ydim, probable_bump_size,
    n, xdim, ydim, rotation_buf[0], scale_buf[0]);
*rotation = rotation_buf[0];
*scale = scale_buf[0];
*present = 1;
//now find precise global alignment parameters
}
else { // send back no go on first detect, then get options for quitting or looking harder
    *present = 0;
}
delete () rotation_buf;
delete () scale_buf;
delete () value;
return(1);
}
int experiment(
    unsigned char *data,
    int n
){
    float *mag = new float(n*n);
    //for(i=0; i<n*n; i++) mag[i]=(float)0.0;
    load_grid_family(); // will immediately return if already done
    reallt2d_in_place(sublimal_grid, 7.0, wr, wi);
    fft2d(sublimal_grid, imag, 7.0, wr, wi);
    return(1);
}
/* main registration program: to be used as main module inside other programs */
int Align::direct_registration {
    int template_xdim,
    int template_ydim,
    unsigned char *suspect,
    int suspect_xdim,
    int suspect_ydim,
    int num_channels
){
    if(1){
        //experiment(ttemplate, template_xdim);
        //return(1);
        int present;
        float rotation, scale;
        extern float *mellin_mag_transform;
        hunt_for_grid(
            suspect,
            suspect_xdim,
            suspect_ydim,
            num_channels,
            1,
            10,
            &present,
            &scale,
            &rotation,
            mellin_mag_transform
        );
        // temporary: place mellin_mag_transform into ttemplate for return
    }
}
// use atemp.bmp as input alignment template file
//
float highest=(float)-1e20, lowest=(float)1e20;
int i, n=128;
for(i=0; i<(n/2+1); i++){
    if( (i/128 < 6) && (abs((i/128)-64) < 6) );
    else {
        if(mellin_mag_transform[i]>highest)mellin_mag_transform[i];
        if(mellin_mag_transform[i]<lowest)lowest=mellin_mag_transform[i];
    }
}
highest = (float)255.0/(highest-lowest);
for(i=0; i<(n/2+1); i++){
    if( (i/128 < 6) && (abs((i/128)-64) < 6) )ttemplate[i] = (unsigned char)100;
    else ttemplate[i] = (unsigned char)( mellin_mag_transform[i] - lowest )*(highest);
}
// use atemp128.bmp as input alignment template file
float highest=(float)-1e20, lowest=(float)1e20;
int i, n=128;
for(i=0; i<(n*n); i++){
    if(mellin_mag_transform[i]>highest)highest=mellin_mag_transform[i];
    if(mellin_mag_transform[i]<lowest)lowest=mellin_mag_transform[i];
}
highest = (float)255.0/(highest-lowest);
for(i=0; i<(n*n); i++){
    ttemplate[i] = (unsigned char)( mellin_mag_transform[i] - lowest )*(highest);
}
}
else {
    int i, fftdim, bits, array_size, lp_array_size;
    int alignment_mode=2, downsample;
    int number_candidates = MAX_CANDIDATES; // number of peaks looked at */
    float rotation[MAX_CANDIDATES], scale[MAX_CANDIDATES], value[MAX_CANDIDATES];
    float x_trans[MAX_CANDIDATES], y_trans[MAX_CANDIDATES], x[5], y[5];
    unsigned char *suspect_lum = new unsigned char(suspect_xdim*suspect_ydim);
    unsigned char *template_lum = new unsigned char(template_xdim*template_ydim);
    // if color image, then create collapse template into a single image.
    // while the real suspect is used during final resampling
    if(num_channels == 3){
        unsigned char *pin,*ptemplate;
        ptemplate = template_lum;
        pin = template+1;
        for(i=0; i<(template_xdim*template_ydim); i++){
            *(ptemplate++) = *pin; // no need for extreme accuracy
            pin+=3;
        }
        ptemplate = suspect_lum;
        pin = suspect+1;
        for(i=0; i<(suspect_xdim*suspect_ydim); i++){
            *(ptemplate++) = *pin; // no need for extreme accuracy
            pin+=3;
        }
    }
    // find working array size after downsampling (if downsampling is called at all)
    fftdim = get_working_dimension(alignment_mode, template_xdim, template_ydim,
        suspect_xdim, suspect_ydim, downsample);
    array_size = fftdim*(fttdim+2);
    lp_array_size = lp_sampling*(lp_sampling+2); // the extra 2 is due to the fft routine
    being used
    bits = (int)( log( double)(fttdim+1) / log( 2.0 ) ); // fftdim should always be power
    of 2
    // create the requisite arrays
    float *template_real = new float(array_size);
    float *template_lp_real = new float(lp_array_size);
    float *suspect_real = new float(array_size);
    float *suspect_lp_real = new float(lp_array_size);
    float *ftemp = new float(array_size);
    float *suspect_copy = new float(array_size);
    // copy the two inputs into the arrays, with any downsampling and windowing applied
    if(num_channels == 1){
        copy_downsample_window(suspect, suspect_xdim, suspect_ydim, suspect_real,
            fttdim, downsample);
        copy_downsample_window(ttemplate, template_xdim, template_ydim, template_real,
            fttdim, downsample);
    }
    else if(num_channels == 3){
        copy_downsample_window(suspect_lum, suspect_xdim, suspect_ydim, suspect_real,
            fttdim, downsample);
        copy_downsample_window(template_lum, template_xdim, template_ydim, template_real,
            fttdim, downsample);
    }
}
}

```

```

        fftdim, downsamples);
    }
    memory(suspect_copy, suspect_real, array_size*sizeof(float) );
    /* real-valued 2D FFT both suspect and template into it's half-plane complex plane */
    realfft2d_in_place(template_real,bits,0,wr,wi);
    realfft2d_in_place(suspect_real,bits,0,wr,wi);

    // calculate fourier mellin transform
    fourier_mellin_transform(template_real,ftemp,fttdim,template_lp_real);
    fourier_mellin_transform(suspect_real,ftemp,fttdim,suspect_lp_real);

    /* assuming the inputs are both real only, then real 2D FFT each */
    realfft2d_in_place(template_lp_real,lp_bits,0,wr,wi);
    realfft2d_in_place(suspect_lp_real,lp_bits,0,wr,wi);

    /* perform generalized matched filter on the two resulting arrays, outputting some number of
    likely candidates, with their associated parameters */
    gmf(template_lp_real,suspect_lp_real,lp_sampling,lp_bits,number_candidates,
        rotation, scale, value, 0);

    // change units on rotation and scale for later stages
    for(i=0;number_candidates;i++){
        rotation[i] = ((float)180.0 / (float)lp_sampling); // converts to degrees
        scale[i] = (float)pow((double)scale_increment,(double)scale[i]); // converts to linear scale
    }

    /* now we have a series of candidates ( or 1, and we just need to get the rotation
    and translation information ) wherein one of them should be
    the correct one; this next routine sifts through all candidates, including both
    the nominal rotation state and the state 180 degrees rotated from the nominal, and
    finds which rotation, scale, and translation gives the highest matched filter
    output; which then will be passed to the last fine tuning stage.
    // Returns best candidate in first element of rotation, scale, x_trans, y_trans
    get_best_candidate(number_candidates,ftemp,fttdim,bits,suspect_copy,
        1+(suspect_xdim-1)/downsample,1+(suspect_ydim-1)/downsample,suspect_xdim,
        suspect_ydim,downsample,rotation,scale,x_trans,y_trans,template_real);

    /* convert the scale/rotation/translation parameters of the downsampled arrays
    into the x and y positions of the four corners of the suspect array, as projected
    onto the template array. Precision in keeping track of the various coordinate systems
    translates into final alignments to well better than a single pixel, especially
    in light of the subtleties involved with downsampling. The four corners
    are labelled 0 through 3 in the arrays x and y, where element 0 is the upper left corner
    of the suspect, element 1 is the upper right, element 2 lower left, element 3 lower right.
    The master 0,0 origin is placed at the upper left of the template array, while
    the centerpoints of the two arrays play a role in rotations. The fifth
    point in the x and y arrays is the centerpoint, used just so you don't have to
    recalculate it all the time.
    get_corners_and_center(x,y,rotation[0],scale[0],x_trans[0],y_trans[0],
        suspect_xdim,suspect_ydim,fttdim,downsample);

    /* now fine tune the result using tricky tricks, see notebook of Nov 28, 1995 */
    if (num_channels == 1){
        for(i=0;i<100;i++){
            fine_tune_x_y(template,suspect_xdim,template_ydim,suspect_xdim,
                suspect_ydim,x,y,rotation);
        }
    }
    else if (num_channels == 3){
        fine_tune_x_y(template_lum,template_xdim,template_ydim,suspect_lum,suspect_xdim,
            suspect_ydim,x,y,rotation);
    }

    /* last but not least, create the output image array, with various options */
    final_image(template,template_xdim,template_ydim,suspect_xdim,
        suspect_ydim,x,y,num_channels,1); // '1' stands for aligned suspect with black everywhere else

    /* Record some results of the alignment process in our status structure */
    m_alignStatus.rotation = rotation[0];
    m_alignStatus.x_scale = scale[0];
    m_alignStatus.y_scale = scale[0];
    m_alignStatus.x_trans = x_trans[0];
    m_alignStatus.y_trans = y_trans[0];

    /* free em all */
    delete (template_real);
    delete (template_lp_real);
    delete (suspect_real);
    delete (suspect_lp_real);
    delete (ftemp);
    delete (suspect_copy);
    delete (suspect_lum);
    delete (template_lum);
}

return(1);
}

// ===== ALIGN_N =====
// FILE: Align_n
// DESCRIPTION:
// Header file for the Alignment core algorithm code and the "Align"
// class used to encapsulate this code.
// The Alignment code is equivalent to Geoff Rhoads "Register" core
// algorithms, which were first created and run as a stand-alone C program
// on the SET, then ported to Win95 and Visual C++ as a "console" program,
// and finally incorporated into the Signer windows application.
// Copyright (C) 1996 Digimarc Incorporated, all rights reserved.
// ===== ALIGN_N =====

```

```

#define ALIGN_H
// A structure used to define results of the alignment process.
typedef struct
{
    float rotation;
    float x_scale;
    float y_scale;
    float x_trans;
    float y_trans;
    float refinement;
} AlignStatus;

// Function prototypes: entry functions
class Align
{
public:
    Align();
    int direct_registration(unsigned char *template,
        int template_xdim,
        int template_ydim,
        unsigned char *suspect,
        int suspect_xdim,
        int suspect_ydim,
        int num_channels);
    // Accessor for status
    const AlignStatus GetAlignStatus(void) const {return m_alignStatus;}

private:
    // Private structure which contains results of alignment
    AlignStatus m_alignStatus;

    int fine_tune_x_y(unsigned char *template,
        int template_xdim,
        int template_ydim,
        unsigned char *suspect,
        int suspect_xdim,
        int suspect_ydim,
        float *x,
        float *y,
        float *rotation);
};

// Function prototypes: private functions
int gmf_ldt(float *real1,
    float *real2,
    float *imaginary1,
    float *imaginary2,
    int dim,
    int bits,
    int *offset);

#endif // ALIGN_H

// AlignDlg.cpp : implementation file
//
#include "stdafx.h"
#include "aligner.h"
#include "AlignDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

// AlignDlg
IMPLEMENT_DYNAMIC(AlignDlg, CFileDialog)

AlignDlg:AlignDlg(BOOL bOpenFileDialog, LPCTSTR lpszDefExt, LPCTSTR lpszFileName,
    DWORD dwFlags, LPCTSTR lpszFilter, CWnd* pParentWnd) :
    CFileDialog(bOpenFileDialog, lpszDefExt, lpszFileName, dwFlags, lpszFilter, pParentWnd)
{
}

BEGIN_MESSAGE_MAP(AlignDlg, CFileDialog)
    //((AFX_MSG_MAP(AlignDlg)
    //((AFX_MSG_MAP(AlignDlg)
    // NOTE - the ClassWizard will add and remove mapping macros here.

```















```

    ij = (i<<nbites)+j ;
    xi = ar[ij] ;
    ar[ij] = ar[ji] ;
    ar[ji] = xi ;
    ar[ji] = ar[ij] ;
    ar[ij] = xi ;
    }
}

fft( &ar[0], &ai[0], nbites, inv, wr, wi, 1 ) ;
for( i = 1 ; i < n ; i++ )
{
    fft( &ar[i<<nbites], &ai[i<<nbites], nbites, inv, wr, wi, 0 ) ;
}

for( i = 1 ; i < n ; i++ )
{
    for( j = 0 ; j < i ; j++ )
    {
        ij = (i<<nbites)+j ;
        ji = (j<<nbites)+i ;
        xi = ar[ij] ;
        ar[ij] = ar[ji] ;
        ar[ji] = xi ;
        ar[ji] = ar[ij] ;
        ar[ij] = xi ;
    }
}

for( i = 0 ; i < n ; i++ )
{
    fft( &ar[i<<nbites], &ai[i<<nbites], nbites, inv, wr, wi, 0 ) ;
}

return(0) ;
}

void reallft_two_arrays(float *array1,float *array2,int nbites,int inv,float *wr,float *wi,int neww)
{
    register int j ;
    register int n ;
    register int nhalf ;
    float temp1[MAX_LINEAR_DIMENSION],temp2[MAX_LINEAR_DIMENSION] ;
    register float *ptemp1 ;
    register float *ptemp2 ;
    register float *par ;
    register float *pai ;
    register float *pai1 ;
    register float *pai2 ;
    register float *ptemp1_1 ;
    register float *ptemp2_1 ;
    n = 1 << nbites ;
    nhalf = n/2 ;
    if(!inv){
        /* sort the results */
        ptemp1 = temp1 ;
        ptemp2 = temp2 ;
        par = array1 ;
        pai = array2 ;
        *ptemp1 = *(par++) ;
        *ptemp2 = *(pai++) ;
        pai1 = &array1[n-1] ;
        pai2 = &array2[n-1] ;
        ptemp1_1 = temp1 ;
        ptemp2_1 = temp2 ;
        for(j=1;j<nhalf;j++){
            *(ptemp1++) = (float)0.5 * (*(par + *par1)) ;
            *(ptemp2++) = (float)0.5 * (*(pai + *pai1)) ;
            *(ptemp1_1++) = (float)0.5 * (*(par - *par1)) ;
            *(ptemp2_1++) = (float)0.5 * (*(pai - *pai1)) ;
            par++ ; par1-- ; pai++ ; pai1-- ;
        }
        temp1[i] = *par ;
        temp2[i] = *pai ;
        /* now copy the results back into original arrays */
        memcpy(array1,temp1,n*sizeof(float)) ;
        memcpy(array2,temp2,n*sizeof(float)) ;
    }
    else {
        /* re-sort results */
        ptemp1 = temp1 ;
        ptemp2 = temp2 ;
        par = array1 ;

```

```

        pai = array2 ;
        *(ptemp1++) = *par ;
        *(ptemp2++) = *pai ;
        par = &array1[n/2] ;
        pai = &array2[n/2] ;
        ptemp1_1 = temp1[n-1] ;
        ptemp2_1 = temp2[n-1] ;
        for(j=1;j<(n/2);j++){
            *(ptemp1++) = (*par - *(pai+1)) ;
            *(ptemp1_1++) = (*par + *(pai+1)) ;
            *(ptemp2++) = (*par+1 + *pai) ;
            *(ptemp2_1++) = (*par+1 - *pai) ;
            par+=2 ;
            pai+=2 ;
        }
        ptemp1 = array1[1] ;
        ptemp2 = array2[1] ;
        fft(array1,array2,nbites,inv,wr,wi,neww) ;
    }
}

/* this routine requires that the input array have two more rows of n appended, into which the
nyquist row will be placed */
int reallft2d_in_place(float *ar,int nbites,int inv,float *wr,float *wi)
{
    register int i ;
    register int j ;
    register int ji ;
    register int j1 ;
    register int n ;
    register int n2 ;
    register int nhalf ;
    register float xr ;
    register float xi ;
    register float x1 ;
    register float x11 ;
    float temp_r[MAX_LINEAR_DIMENSION],temp_i[MAX_LINEAR_DIMENSION] ;
    register float *ptemp_r ;
    register float *ptemp_i ;
    register float *par ;
    register float *pai ;
    register float *pai1 ;
    register float *pai2 ;
    register float *ptemp_r1 ;
    register float *ptemp_i1 ;
    n = 1 << nbites ;
    n2 = n*2 ;
    nhalf = n/2 ;
    if (!inv){
        /* pre-transpose */
        for( i = 1 ; i < n ; i++ )
        {
            for( j = 0 ; j < i ; j++ )
            {
                ij = (i<<nbites)+j ;
                ji = (j<<nbites)+i ;
                xr = ar[ij] ;
                ar[ij] = ar[ji] ;
                ar[ji] = xr ;
            }
        }
        for( i = 0 ; i < nhalf ; i++ )
        {
            if(i==0)fft( &ar[0], &ar[n], nbites, inv, wr, wi, 1 ) ;
            else fft( &ar[n2+i], &ar[n2+i+n], nbites, inv, wr, wi, 0 ) ;
        }
        /* sort and pack results */
        ptemp_r = temp_r ;
        ptemp_i = temp_i ;
        ptemp_1 = &temp_i[2] ;
        par = &ar[n2+1] ;
        pai = &ar[n2+i+n] ;
        *(ptemp_r++) = *(par++) ;
        *(ptemp_i++) = *(pai--),
        pai = &ar[n2+i+n] ;
        for(j=1;j<nhalf;j++){
            *(ptemp_r++) = (float)0.5 * (*(par + *par1)) ;
            *(ptemp_i++) = (float)0.5 * (*(pai + *pai1)) ;
            *(ptemp_r_1++) = (float)0.5 * (*(par - *par1)) ;
            *(ptemp_i_1++) = (float)0.5 * (*(pai - *pai1)) ;
            par++ ; par1-- ; pai++ ; pai1-- ;
        }
        temp_i[0] = *par ;

```

```

temp_i[1] = *pai;

/* now copy the results back into original arrays */
memcpy(kar[n2*1], temp_r, n*sizeof(float));
memcpy(kar[n2*1+n], temp_i, n*sizeof(float));

/* transpose */
for( i = 2; i < n; i+=2 ) {
    for( j = 0; j < i; j+=2 ) {
        ij = (i<nbits)*j;
        ji = (j<nbits)*i;
        xr = ar[ij];
        xi = ar[ij+n];
        xrl = ar[ij+1];
        xil = ar[ij+1+n];
        ar[ij] = ar[ji];
        ar[ij+n] = ar[ji+n];
        ar[ij+1] = ar[ji+1];
        ar[ij+1+n] = ar[ji+1+n];
        ar[ji] = xr;
        ar[ji+n] = xi;
        ar[ji+1] = xrl;
        ar[ji+1+n] = xil;
    }
}

/* place nyquist row into n*n row, and zero out their imaginary rows */
memcpy(kar[n*n], kar[n], n*sizeof(float));
memset(kar[n], 0, n*sizeof(float));
memset(kar[n*n], 0, n*sizeof(float));

for( i = 0; i < nhalf+1; i++ ) fft( kar[n2*i], kar[n2*i+n], nbits, inv, wr, wi, 0 );

/* finally, shift the arrays in order to simplify external processing */
for( i=0; i<n2; i++ ) {
    memcpy(temp_r, kar[i*n], nhalf*sizeof(float));
    memcpy(kar[i*n], kar[nhalf+i*n], nhalf*sizeof(float));
    memcpy(kar[nhalf+i*n], temp_r, nhalf*sizeof(float));
}

else {
    /* undo format */
    for( i=0; i<(n+2); i++ ) {
        memcpy(temp_r, kar[i*n], (n/2)*sizeof(float));
        memcpy(kar[i*n], kar[n/2+i*n], (n/2)*sizeof(float));
        memcpy(kar[n/2+i*n], temp_r, (n/2)*sizeof(float));
    }

    fft( kar[0], kar[n], nbits, inv, wr, wi, 1 );
    for( i = 1; i < (n+2)/2; i++ ) fft( kar[(2*i)*n], kar[(2*i+1)*n], nbits, inv, wr, wi, 0 );
    memcpy(kar[n], kar[n*n], n*sizeof(float));

    /* transpose */
    for( i = 2; i < n; i+=2 ) {
        for( j = 0; j < i; j+=2 ) {
            ij = (i<nbits)*j;
            ji = (j<nbits)*i;
            xr = ar[ij];
            xi = ar[ij+n];
            xrl = ar[ij+1];
            xil = ar[ij+1+n];
            ar[ij] = ar[ji];
            ar[ij+n] = ar[ji+n];
            ar[ij+1] = ar[ji+1];
            ar[ij+1+n] = ar[ji+1+n];
            ar[ji] = xr;
            ar[ji+n] = xi;
            ar[ji+1] = xrl;
            ar[ji+1+n] = xil;
        }
    }

    /* re-sort results */
    temp_r = temp_r;
    temp_i = temp_i;
    par = kar[(2*1)*n];
    *ptemp_r++ = *par++;
    *ptemp_i++ = *par++;

    pai = kar[(2*1+1)*n];
    temp_r1 = temp_r[n-1];
    temp_i1 = temp_i[n-1];
    for( j=1; j<(n/2); j++ ) {
        *ptemp_r++ = *par * (pai+1);
        *ptemp_r1-- = (*par * (pai+1));
    }
}

```

# PPT.B

```

/* FILE: PPT.h
*
* DESCRIPTION:
* Include file for Geoff's PPT routines. Callers of the PPT functions
* should include this header file to pick up the function prototypes.
*
* Copyright (C) Digimarc Corporation, 1996. All rights reserved.
*
* void fft(float *a,
*          int nbits,
*          int inv,
*          float *wr,
*          float *wi,
*          int neww);
/* the real part of the array */
/* log base 2 of the number of elements in the arrays */
/* nonzero to indicate the inverse transform */
/* the real part of an array of coefficients */
/* the imag part of an array of coefficients */
/* nonzero to indicate the coefficients must be calced */

int fft2d(float *a, float *ai, int nbits, int inv, float *wr, float *wi);

void realfft_two_arrays(float *array1, float *array2,
                        int nbits, int inv, float *wr, float *wi, int neww);

int realfft2d_in_place(float *a, int nbits, int inv, float *wr, float *wi);

// File: Image.cpp
//
// Contains the implementation for the Image class. Image objects
// are used to contain the image data, and provide a more convenient
// set of services related to accessing the image data as well as
// attribute variables describing the image.
//
// #include "Image.h"
// #include "dibapi.h"
// #include "stdafx.h"
//
// Image(HDIB HDIB)
//
// Constructor which creates an Image object, given a handle to
// a DIB which is already in memory.
//
// Image::Image(HDIB HDIB)
// {
//     BITMAPINFO *bmi_info;
//     m_hpPackedData = NULL;
//     m_fileOK = TRUE;
//     // its already been opened.
// }

```

# IMAG.BT.CPP

```

// File: Image.cpp
//
// Contains the implementation for the Image class. Image objects
// are used to contain the image data, and provide a more convenient
// set of services related to accessing the image data as well as
// attribute variables describing the image.
//
// #include "Image.h"
// #include "dibapi.h"
// #include "stdafx.h"
//
// Image(HDIB HDIB)
//
// Constructor which creates an Image object, given a handle to
// a DIB which is already in memory.
//
// Image::Image(HDIB HDIB)
// {
//     BITMAPINFO *bmi_info;
//     m_hpPackedData = NULL;
//     m_fileOK = TRUE;
//     // its already been opened.
// }

```

```

m_hDIB = hDIB;
m_lpDIB = (LPSTR) ::GlobalLock( (HGLOBAL) m_hDIB);
// NOTE: THE FOLLOWING MEMBER POINTERS ARE ONLY VALID WHILE
// WE KEEP THE DIB DATA LOCKED IN MEMORY. FOR THIS IMPLEMENTATION,
// I LEAVE THE DATA LOCKED UNTIL THE OBJECT IS DESTROYED.

bmi_info = (BITMAPINFO *) m_lpDIB;
// Set up a pointer to the BITMAPINFOHEADER and RGBQUAD array.
m_lpBmiHeader = &bmi_info->bmiHeader;
m_lpBmiColors = &bmi_info->bmiColors[0];

// Set the pointer to the image data.
m_hpDIBBits = (unsigned char *) ::FindDIBBits(m_lpDIB);

m_BitsPerPixel = m_lpBmiHeader->biBitCount;
m_XDim = m_lpBmiHeader->biWidth;
m_YDim = m_lpBmiHeader->biHeight;
m_Compression = m_lpBmiHeader->biCompression;
m_WidthInBytes = WIDTHBYTES(m_XDim * m_BitsPerPixel);
}

// Image (HDIIB hDIB)
// Constructor which creates an Image object, given the name of a DIB
// or BMP file.
// Image::Image(CString filename)
{
    CFile file;
    CFileException fe;
    BITMAPINFO *bmi_info;
    m_hpPackedData = NULL;

    if (!file.Open(filename, CFile::modeRead | CFile::shareDenyWrite, &fe))
    {
        CString msg("Error reading image file: ");
        msg += filename;
        MessageBox(NULL, msg, NULL, MB_ICONINFORMATION | MB_OK);
        m_fileOK = FALSE;
    }
    else
        m_fileOK = TRUE;

    // Try to read the DIB file, catch any exceptions.
    TRY
    {
        m_hDIB = ::ReadDIBFile(file);
    }
    CATCH(CFileException, eLoad)
    {
        file.Abort();
        MessageBox(NULL, "Error reading the image file", NULL,
        MessageBox(NULL, MB_ICONINFORMATION | MB_OK);
        m_hDIB = NULL;
        m_fileOK = FALSE;
    }
    END_CATCH

    m_lpDIB = (LPSTR) ::GlobalLock( (HGLOBAL) m_hDIB);
    // NOTE: THE FOLLOWING MEMBER POINTERS ARE ONLY VALID WHILE
    // WE KEEP THE DIB DATA LOCKED IN MEMORY. FOR THIS IMPLEMENTATION,
    // I LEAVE THE DATA LOCKED UNTIL THE OBJECT IS DESTROYED.

    bmi_info = (BITMAPINFO *) m_lpDIB;
    // Set up a pointer to the BITMAPINFOHEADER and RGBQUAD array.
    m_lpBmiHeader = &bmi_info->bmiHeader;
    m_lpBmiColors = &bmi_info->bmiColors[0];

    // Set the pointer to the image data.
    m_hpDIBBits = (unsigned char *) ::FindDIBBits(m_lpDIB);
    m_BitsPerPixel = m_lpBmiHeader->biBitCount;
    m_XDim = m_lpBmiHeader->biWidth;
    m_YDim = m_lpBmiHeader->biHeight;
    m_Compression = m_lpBmiHeader->biCompression;
    m_WidthInBytes = WIDTHBYTES(m_XDim * m_BitsPerPixel);
}

// -Image()
}

// The destructor for the Image class of objects.
// ~Image()
{
    ::GlobalUnlock( (HGLOBAL) m_hDIB);
    if (m_hpPackedData != NULL)
    {
        ::GlobalUnlock( (HGLOBAL) m_hpPackedData);
        ::GlobalFree( (HGLOBAL) m_hpPackedData);
    }
}

// ~MakePackedData()
// This function copies the DIB image data into a packed format. This
// is important for two reasons: 1) the DIB formatted data is arranged
// so that each scan line starts on a long word boundary, so there may
// be up to 3 unused bytes at the end of each scan line in the case of
// 8 bit data. This arrangement is inconvenient when passing the image
// data to the core algorithms. Also, 2) if a palette is being used
// up the actual image values using the palette and places these values
// in the packed data array. The member variable m_hpPackedData is the
// handle to the packed data.
// WARNING: CURRENT IMPLEMENTATION ASSUMES 8 BIT GRAY-SCALE IMAGE DATA
// void Image::MakePackedData(void)
{
    unsigned char *hpline;
    unsigned char *hpdata;
    int line_cnt, line, i;
    BOOLEAN bottom_up;

    // Create space and get handle for the packed data of the image.
    m_hpPackedData = ::GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT,
    if (m_hpPackedData == 0)
        AfxThrowMemoryException();

    // Lock the packed data global memory (leave locked until destructor).
    m_hpPackedData = (unsigned char *) ::GlobalLock( (HGLOBAL) m_hpPackedData);

    hpdata = m_hpPackedData;

    // Image may be top to bottom or bottom to top.
    if (m_lpBmiHeader->biHeight > 0)
    {
        bottom_up = TRUE;
        line = m_YDim - 1;
    }
    else
    {
        bottom_up = FALSE;
        line = 0;
    }

    // TEST CODES
    // For Geoff, don't let it correct for bottom_up
    bottom_up = FALSE;
    line = 0;

    // Now go through each line and create the packed array.
    for (line_cnt = 0; line_cnt < m_YDim; line_cnt++)
    {
        // Set pointer to first byte for this scan line.
        hpline = &m_hpDIBBits[line * (long) m_WidthInBytes];
        for (i = 0; i < m_XDim; i++)
        {
            if (m_BitsPerPixel == 24)
                *hpdata++ = hpline[i];
            else
            {
                // For 8 bit (and any other non 24 bit data) we
                // take the image data to be indicos into the color
                // table. We look up the actual value. Note we
                // assume gray-scale (i.e., r,g,b tripled are all equal -
                // we read the green.
                *hpdata++ = m_lpBmiColors[hpline[i]].rgbGreen;
            }
        }
        if (bottom_up) line--;
        else line++;
    }
}

```

```

////////////////////////////////////
// UnpackData()
//
// This function moves the contents of the packed data array back into
// the DIB data space. This would be used, for example, after one the
// core algorithms have been used to sign the data in the packed array,
// and we want to update the DIB to reflect the changes. Note that this
// requires that we create our own palette, since otherwise we don't know
// that the new data values have corresponding entries in the palette.
//
// WARNING: CURRENT IMPLEMENTATION ASSUMES 8 BIT GRAY-SCALE IMAGE DATA
////////////////////////////////////
void Image::UnpackData(void)
{
    unsigned char *hplLine;
    unsigned char *hpData;
    int line_cnt, line, i;
    BOOL bLineUp, bottom_up;

    // Image may be top to bottom or bottom to top.
    if (m_lpBmiHeader->biHeight > 0)
    {
        bottom_up = TRUE;
        line = m_YDim - 1;
    }
    else
    {
        bottom_up = FALSE;
        line = 0;
    }

    // TEST CODE
    // For Geoff: don't let it correct for bottom_up
    bottom_up = FALSE;
    line = 0;

    hpData = m_hpPackedData;
    for (line_cnt = 0; line_cnt < m_YDim; line_cnt++)
    {
        // Set pointer to first byte for this scan line.
        hplLine = &m_hpDIBbits[line * (long)m_WidthInBytes];
        for (i = 0; i < m_XDim; i++)
        {
            hplLine[i] = *hpData++;
        }
        if (bottom_up)
            else line++;
    }

    // Next, we force the palette to be our standard 8 bit grey-scale
    // palette.
    if (m_BitsPerPixel == 8)
    {
        // Set ptr to beginning of palette
        LPGRAYQUAD pal = m_lpBmiColors;

        for (i = 0; i < 256; i++)
        {
            pal[i].rgbBlue = pal[i].rgbGreen = pal[i].rgbRed = i;
        }
    }
    else
    {
        MessageBox(NULL, "Can only unpack 8 bit image data", NULL,
            MB_ICONEXCLAMATION | MB_OK);
    }
}

////////////////////////////////////
// File: Image.cpp
//
// Contains the implementation for the Image class. Image objects
// are used to contain the image data, and provide a more convenient
// set of services related to accessing the image data as well as
// attribute variables describing the image.
//
// #include "Image.h"
// #include "dibapi.h"
// #include "stdafx.h"
//
// Image(HDIB hDIB)
//
// Constructor which creates an Image object, given a handle to
// a DIB which is already in memory.
////////////////////////////////////
Image::Image(HDIB hDIB)
{
    // Create a pointer to the DIB info.
    // its already been opened.
    m_hpPackedData = NULL;
    m_fileOK = TRUE;
    m_hDIB = hDIB;

    m_lpDIB = (LPSTR) ::GlobalLock( (HGLOBAL) m_hDIB);

    // NOTE: THE FOLLOWING MEMBER POINTERS ARE ONLY VALID WHILE
    // WE KEEP THE DIB DATA LOCKED IN MEMORY. FOR THIS IMPLEMENTATION,
    // I LEAVE THE DATA LOCKED UNTIL THE OBJECT IS DESTROYED.

    bmi_info = (BITMAPINFO *) m_lpDIB;
    // Set up a pointer to the BITMAPINFOHEADER and RGBQUAD array.
    m_lpBmiHeader = &bmi_info->bmiHeader;
    m_lpBmiColors = &bmi_info->bmiColors[0]; // will be null for 24 bit
    // Set the pointer to the image data
    m_hpDIBbits = (unsigned char *) ::FindDIBbits(m_lpDIB);

    m_BitsPerPixel = m_lpBmiHeader->biBitCount;
    m_XDim = m_lpBmiHeader->biWidth;
    m_YDim = m_lpBmiHeader->biHeight;
    m_Compression = m_lpBmiHeader->biCompression;
    m_WidthInBytes = WIDTHBYTES(m_XDim * m_BitsPerPixel);
}

////////////////////////////////////
// Image(HDIB hDIB)
//
// Constructor which creates an Image object, given the name of a DIB
// or BMP file.
//
// Image(CString filename)
{
    CFile file;
    CFileException fe;
    BITMAPINFO *bmi_info;
    m_hpPackedData = NULL;

    if (!file.Open(filename, CFile::modeRead | CFile::shareDenyWrite, &fe))
    {
        CString msg("Error reading image file: ");
        msg += filename;
        MessageBox(NULL, msg, NULL, MB_ICONINFORMATION | MB_OK);
        m_fileOK = FALSE;
    }
    else
    {
        m_fileOK = TRUE;
    }

    // Try to read the DIB file, catch any exceptions.
    TRY
    {
        m_hDIB = ::ReadDIBFile(file);
    }
    CATCH(CFileException, eLoad)
    {
        file.Abort();
        MessageBox(NULL, "Error reading the image file", NULL,
            m_hDIB = NULL;
            m_fileOK = FALSE;
        }
    }
    END_CATCH

    m_lpDIB = (LPSTR) ::GlobalLock( (HGLOBAL) m_hDIB);

    // NOTE: THE FOLLOWING MEMBER POINTERS ARE ONLY VALID WHILE
    // WE KEEP THE DIB DATA LOCKED IN MEMORY. FOR THIS IMPLEMENTATION,
    // I LEAVE THE DATA LOCKED UNTIL THE OBJECT IS DESTROYED.

    bmi_info = (BITMAPINFO *) m_lpDIB;
    // Set up a pointer to the BITMAPINFOHEADER and RGBQUAD array.
    m_lpBmiHeader = &bmi_info->bmiHeader;
    m_lpBmiColors = &bmi_info->bmiColors[0];
    // Set the pointer to the image data.
    m_hpDIBbits = (unsigned char *) ::FindDIBbits(m_lpDIB);
    m_BitsPerPixel = m_lpBmiHeader->biBitCount;
    m_XDim = m_lpBmiHeader->biWidth;
}

```



```

m_YDim = m_lpBmiHeader->biHeight;
m_Compression = m_lpBmiHeader->biCompression;
m_WidthInBytes = WIDTHBYTES(m_XDim * m_BitsPerPixel);
}

//=====
// The destructor for the Image class of objects.
Image::~Image(void)
{
    ::GlobalUnlock( (HGLOBAL) m_hDIB );
    if (m_hpPackedData != NULL)
    {
        ::GlobalUnlock( (HGLOBAL) m_hpPackedData );
    }
}

//=====
// This function copies the DIB image data into a packed format. This
// is important for two reasons: 1) the DIB formatted data is arranged
// so that each scan line starts on a long word boundary, so there may
// be up to 3 unused bytes at the end of each scan line in the case of
// 8 bit data. This arrangement is inconvenient when passing the image
// data to the core algorithms. Also, 2) if a palette is being used
// (this is the case for all but 24 bit image data), this routine looks
// up the actual image values using the palette and places these values
// in the packed data array. The member variable m_hpackedData is the
// handle to the packed data.
// The force_to_1_chan argument is an optional boolean. It defaults
// to FALSE (see function prototype in Image.h). If set to TRUE,
// only 1 channel of packed data is created, even if the image is 3
// channels. This is useful when creating snowy images from RGB
// images, since we currently always want 1 channel snowy images.
void Image::MakePackedData(BOOLEAN force_to_1_chan)
{
    unsigned char *hpLine;
    unsigned char *hpData;
    int line_cnt, line, i, j;
    long size;
    BOOLEAN bottom_up;

    // Create space and get handle for the packed data of the image.
    size = m_XDim * m_YDim;
    // For 24 bit true color, we will pack R,G,B values, so triple the size.
    if (m_BitsPerPixel == 24 && force_to_1_chan == FALSE)
        size *= 3;
    m_hpackedData = ::GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT, size);
    if (m_hpackedData == 0)
        AfxThrowMemoryException();

    // Lock the packed data global memory (leave locked until destructor).
    m_hpPackedData = (unsigned char *)::GlobalLock( (HGLOBAL) m_hpackedData );
    hpData = m_hpPackedData;

    // Image may be top to bottom or bottom to top.
    if (m_lpBmiHeader->biHeight > 0)
    {
        bottom_up = TRUE;
        line = m_YDim - 1;
    }
    else
    {
        bottom_up = FALSE;
        line = 0;
    }

    // TEST CODE
    // For Geoff: don't let it correct for bottom_up
    // bottom_up = FALSE;
    // line = 0;

    hpData = m_hpPackedData;
    for (line_cnt = 0; line_cnt < m_YDim; line_cnt++)
    {
        // Set pointer to first byte for this scan line.
        hpLine = &m_hpDIBbits[line * (long) m_WidthInBytes];
        for (i = 0, j = 0; i < m_XDim; i++)
        {
            if (m_BitsPerPixel == 24)
            {
                hpLine[j+2] = *hpData++; // red
                hpLine[j+1] = *hpData++; // green
                hpLine[j+0] = *hpData++; // blue
                j += 3;
            }
            else
            {
                *hpData++ = hpLine[j+1]; // take just green to convert
                // to 1 channel data.
                j += 3;
            }
            // For 8 bit (and any other non 24 bit data) we
            // take the image data to be indices into the color
            // table. We look up the actual value. Note we
            // assume grey-scale (i.e., r,g,b triples are all equal -
            // we read the green.
            *hpData++ = m_lpBmiColors[hpLine[i]].rgbGreen;
        }
        if (bottom_up) line--;
        else line++;
    }

    //=====
    // UnpackData()
    // This function moves the contents of the packed data array back into
    // the DIB data space. This would be used, for example, after one the
    // core algorithms have been used to sign the data in the packed array.
    // and we want to update the DIB to reflect the changes. Note that this
    // requires that we create our own palette, since otherwise we don't know
    // that the new data values have corresponding entries in the palette.
    // WARNING: CURRENT IMPLEMENTATION ASSUMES 8 BIT GRAY-SCALE IMAGE DATA
    // OR 24 BIT COLOR IMAGES DATA
    //=====
    void Image::UnpackData(void)
    {
        unsigned char *hpLine;
        unsigned char *hpData;
        int line_cnt, line, i, j;
        BOOLEAN bottom_up;

        // Image may be top to bottom or bottom to top.
        if (m_lpBmiHeader->biHeight > 0)
        {
            bottom_up = TRUE;
            line = m_YDim - 1;
        }
        else
        {
            bottom_up = FALSE;
            line = 0;
        }

        // TEST CODE
        // For Geoff: don't let it correct for bottom_up
        // bottom_up = FALSE;
        // line = 0;

        hpData = m_hpPackedData;
        for (line_cnt = 0; line_cnt < m_YDim; line_cnt++)
        {
            // Set pointer to first byte for this scan line.
            hpLine = &m_hpDIBbits[line * (long) m_WidthInBytes];
            for (i = 0, j = 0; i < m_XDim; i++)
            {
                if (m_BitsPerPixel == 24)
                {
                    hpLine[j+2] = *hpData++; // red
                    hpLine[j+1] = *hpData++; // green
                    hpLine[j+0] = *hpData++; // blue
                    j += 3;
                }
                else
                {
                    hpLine[i] = *hpData++;
                }
                if (bottom_up) line--;
                else line++;
            }
        }

        // Next, we force the palette to be our standard 8 bit grey-scale
        // palette.

```

```

if (m_bitsPerPixel == 8)
{
    // Set ptr to beginning of palette
    LPBQUAD pal = m_lpBmiColors;
    for (i = 0; i < 256; i++)
    {
        pal[i].rgbBlue = pal[i].rgbGreen = pal[i].rgbRed = i;
    }
}
else if (m_bitsPerPixel == 24)
{
    // Don't do any palette work for 24 bit color: there is no palette.
}
else
{
    MessageBox(NULL, "Can only unpack 8 and 24 bit image data", NULL,
        MB_ICONEXCLAMATION | MB_OK);
}
}

// .....
// FILE: Image.h
// .....
// DESCRIPTION:
// * The Image class is used to read .BMP and .DIB image files, and
// * manage an internal representation of them in memory. The goal is
// * to provide a set of service which insulate the caller from having to
// * deal with the specifics of the DIB format. Also, the approach tends
// * to isolate platform specific and file format specific details to this
// * class. For example, adding support for a different type of file
// * format would affect this class, but not the callers.
// * This header file should be included by any module which creates or
// * makes use of Image objects.
// * CREATION DATE: September 5, 1995
// * Copyright (c) 1995 Digimarc Incorporated, all rights reserved.
// * .....
// #ifndef IMAGE_H
// #define IMAGE_H
// #include "stdafx.h"
// #include "dibapi.h"

class Image
{
public:
    // Constructors...
    Image(HDIB hDIB); // Takes a handle to a loaded DIB
    Image(const CString filename); // Takes a filename
    ~Image(void);
    void Image::MakePackedData(void);
    void Image::MakePackedData(BOOLEAN force_to_1_chan = FALSE);
    void Image::UnpackData();

    // Accessors:
    HDIB GetHDIB(void) {return m_hDIB;}
    LPSTR GetLPDIB(void) {return m_lpDIB;}
    BITMAPINFOHEADER *GetBmiHdr(void) {return m_lpBmiHeader;}
    RGBQUAD *GetPalette(void) {return m_lpBmiColors;}
    unsigned char *GetDIBData(void) {return m_hDIBData;}
    unsigned char *GetPackedData(void) {return m_hPackedData;}
    int GetBitsPerPixel(void) {return m_bitsPerPixel;}
    WORD GetSizeOfPalette(void) {return ::PaletteSize(m_lpDIB);}
    WORD GetSizeOfHeader(void) {return ::PaletteSize(m_lpDIB);}
    WORD GetNumColors(void) {return ::DIBNumColors(m_lpDIB);}
    LONG GetXDim(void) {return m_XDim;}
    LONG GetYDim(void) {return m_YDim;}
    BOOL GetFileOK(void) {return m_fileOK;}

    // Private member functions
private:
    // Handle to the DIB.
    HDIB m_hDIB;
    LPSTR m_lpDIB; // Pointer to top of DIB, locked in memory
    // Pointers to the bitmap info header structure, and the palette array.
    LPBITMAPINFOHEADER m_lpBmiHeader; // Points to header structure
    FAR *m_lpBmiColors; // Ptr to beginning of palette array
    *m_hDIBData; // Pointer to DIB bits
    *m_hPackedData; // Handle for the packed data space
    unsigned char m_hPackedData; // Pointer to Packed copy of data.
    LONG m_XDim; // X dimension of image (number of lines)
    LONG m_YDim; // Y dimension of image (number of lines)
    int m_bitsPerPixel;
    LONG m_WidthInBytes;
    DWORD m_Compression;
    BOOL m_fileOK;
};

// #endif // IMAGE_H

// .....
// mainfrm.cpp : implementation of the CMainFrame class
// .....
// #include "stdafx.h"
// #include "signer.h"
// #include "mainfrm.h"
// #ifdef _DEBUG
// #undef THIS_FILE
// static char BASED_CODE THIS_FILE[] = __FILE__;
// #endif
// .....
// CMainFrame
// IMPLEMENT_DYNAMIC(CMainFrame, CWnd)
// BEGIN_MESSAGE_MAP(CMainFrame, CWnd)
//     ON_WM_CREATE()
//     ON_WM_PALETTECHANGED()
//     ON_WM_QUEYRNPALLETTE()
//     ON_WM_QUIT()
//     ON_MESSAGE_MAP()
// END_MESSAGE_MAP()
// .....
// arrays of IDs used to initialize control bars
// .....
// toolbar buttons - IDs are command buttons
// static UINT based_code buttons[] =
// {
//     // same order as in the bitmap 'toolbar.bmp'
//     ID_FILE_NEW,
//     ID_FILE_OPEN,
//     ID_FILE_SAVE_AS,
//     ID_SEPARATOR,
//     ID_EDIT_COPY,
//     ID_EDIT_PASTE,
//     ID_SEPARATOR,
//     ID_FILE_PRINT,
//     ID_APP_ABOUT,
// };
// static UINT based_code indicators[] =
// {
//     ID_SEPARATOR, // status line indicator
//     ID_INDICATOR_CAPS,
//     ID_INDICATOR_NUM,
//     ID_INDICATOR_SCLK,
// };
// .....
// CMainFrame construction/destruction
// .....
// CMainFrame::CMainFrame()
// {
//     CMainFrame::~CMainFrame()
// }
// .....
// int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)

```









```

break;
case slash:
    m_recoveredAsciiMsg(i) = '/';
break;
case backslash:
    m_recoveredAsciiMsg(i) = '\\';
break;
default:
    m_recoveredAsciiMsg(i) = '?'; // When we don't recognize the character.
break;
}

// Add a Null terminator
m_recoveredAsciiMsg(m_msgLength) = '\0';

// Compute the checksum of the read message
m_computedReaderChecksum = ComputeChecksum(m_compactMsg, m_msgLength);

////////////////////
// ComputeChecksum()
// This function is passed a pointer to the compact message
// string containing a message. It computes and returns the checksum.
// The checksum algorithm used is a simple "spiral add", and the
// size of the checksum is PACKED_BITS_PER_CHAR (although it is
// stored as an unsigned char).
//
// NOTE:
// There is an implicit assumption that PACKED_BITS_PER_CHAR < 8
// If this changes, mods will be needed in this code.
//
// unsigned char PackedMsg::ComputeChecksum(char *pMsg, int length)
{
    int
    i;
    unsigned char
    csu = 0;
    const unsigned char carry_bit_mask = (1 << PACKED_BITS_PER_CHAR);
    const unsigned char remove_carry_bit_mask = ~carry_bit_mask;

    for (i = 0; i < length; i++)
    {
        // Rotate the checksum: shift left and OR in the carry bit.
        csu = csu << 1;
        if (csu & carry_bit_mask)
        {
            csu |= 1;
            csu ^= remove_carry_bit_mask;
        }

        // Add the next character
        csu += (unsigned char) *pMsg;

        // We want an unsigned add of length PACKED_BITS_PER_CHAR,
        // so remove the carry bit if its there.
        csu ^= remove_carry_bit_mask;

        pMsg++;
    }

    return csu;
}

////////////////////////////////////
// FILE: PackMsg.h
//
// DESCRIPTION:
// The PackMsg class is responsible for creating an efficient binary
// coding representation of the ASCII message the user wishes to embed
// in the image. This representation is "efficient" in that it packs
// the message into a format which requires fewer total bits than that
// used by the equivalent ASCII representation.
//
// This header file should be included by any module which creates or
// makes use of PackMsg objects.
//
// CREATION DATE: August 16, 1995
//
// Copyright (c) 1995 Digimarc Incorporated, all rights reserved.
//
// \.....
// #ifndef PACKMSG_H
// #define PACKMSG_H

```

```

// #include "digimarc.h"
// #include "Params.h"

// #define PACKED_BITS_PER_CHAR 6 // We will use 6 bits per user character

// We're going to use a 6 bit representation of up to 64 alphanumeric
// plus special characters. The following enumeration indicates how
// each will be represented. There first item takes value 0, 2nd item
// takes 1, ....
enum PackedChar
{
    zero, one, two, three, four, five, six, seven, eight, nine,
    A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,
    space, period, comma, colon, slash, backslash,
    undefined;
};

typedef char * Compact_Msg;

class PackedMsg
{
public:
    // Constructor: takes user's input message and creates the packed version.
    PackedMsg(const char *user_msg);

    // A Constructor for use by the reader.
    PackedMsg(int msg_length);

    // An accessor allows callers read-only access to the packed msg.
    const Compact_Msg getCompactMsg(void) const;
    int getCompactMsgSize(void) const;
    unsigned char *getMsgBitArray(void) const {return m_msgBitArray;}
    int getMsgBitArrayLength(void) const {return m_msgBitArrayLength;}
    char *getAsciiMsg(void) const {return m_asciiMsg;}
    unsigned char *getReaderBitArray(void) const {return m_readerBitArray;}
    char *getRecoveredAsciiMsg(void) const {return m_recoveredAsciiMsg;}

    int GetNumCorrectBits(void) const {return m_correctBits;}
    float GetPercentCorrect(void) const
    {
        return (float) m_correctBits * (float)100.0 / (float) m_msgBitArrayLength;
    }

    // Checksum accessors.
    unsigned char GetSignerChecksum(void) {return m_checksum;}
    unsigned char GetReaderChecksum(void) {return m_recoveredChecksum;}
    unsigned char GetComputedReaderChecksum(void) {return m_computedReaderChecksum;}

    int GetMsgLength(void) const {return m_msgLength;}

    // Function to unpack a message, for use by the recognizer...
    void BitsToString(void);

    // Destructor
    ~PackedMsg(void);

private:
    // Private member functions
    void PackMessage(void);
    unsigned char ComputeChecksum(char *pMsg, int length);

private:
    // Private data
    char
    *m_asciiMsg; // The original ASCII message ASCII null terminated.
    int
    m_msgLength; // No. of chars (not included null terminator.
    Compact_Msg
    m_compactMsg; // The message in the packed format.

    unsigned char
    *m_msgBitArray; // Core signer algorithm wants one bit per char.
    int
    m_msgBitArrayLength; // Includes checksum.
    unsigned char
    *m_readerBitArray; // Array of bits recovered by reader,
    // includes checksum.
    char
    *m_recoveredAsciiMsg; // The recovered message
    unsigned char
    m_checksum;
    unsigned char
    m_recoveredChecksum;
    unsigned char
    m_computedReaderChecksum;

    int
    m_correctBits;
};

#endif // PACKMSG_H

```

PARAMS\_CPP

```

* FILS: Paramo.cpp
*
* DESCRIPTION:
* Implementation of the Parameters classes: SignerParams and
* ReaderParams.
*
* CREATION DATE: September 8, 1995
* Copyright (c) 1995 Digimarc Incorporated, all rights reserved.
* .....
#include "params.h"
#include "scdata.h"
#include "string.h"
#include "strutree.h"

// CONSTRUCTOR FOR SIGNER PARAMS OBJECT WHICH
// TAKES THIS COMMAND LINE STRING AS AN ARGUMENT.
// .....

SignerParams::SignerParams(LPSTR cmd_line)
{
    char *dash_ptr, *cmd_type, *cmd, *commands;
    const char *dbg_msg_ptr;

    parameters.input_filename = NULL;
    parameters.message = "Default Message";
    parameters.output_filename = NULL;
    parameters.registry_name = NULL;

    parameters.user_key = 1;
    parameters.gain = (float) 100.0;
    parameters.gamma = (float) 0.07;
    parameters.bump_size = 1;
    parameters.lut_scale = (float) 100.0;
    parameters.super_reader_flag = FALSE;
    dbg_msg_ptr = (const char *) GetMessage();
    TRACE("Debug in SignerParams constructor. Message is: %s\n", dbg_msg_ptr);

    // Make a copy of the command line that we can mutilate
    commands = new char[strlen(cmd_line) + 1];
    strcpy(commands, cmd_line);
    dash_ptr = NULL;

    // If the command line doesn't start w/ a '.', then the command line is
    // a single argument: the filename. This case comes up when the program
    // is invoked by dragging a filename onto the executable in Win95 explorer.
    if (strlen(cmd_line) > 0 && cmd_line[0] != '.')
    {
        parameters.input_filename = new char[strlen(cmd_line) + 1];
        strcpy(parameters.input_filename, cmd_line);
    }
    // Otherwise, we check for the multiple argument format of the command line,
    // in which arguments pairs are used, e.g., "-f <filename>".
    else
    {
        do
        {
            // Find the last '.' character
            dash_ptr = strrchr(cmd_line, '.');
            if (dash_ptr != NULL)
            {
                cmd_type = dash_ptr + 1;
                cmd = cmd_type + 1;

                // Create an in-core input stream
                istrstream istrstream(cmd, strlen(cmd));

                switch (*cmd_type)
                {
                    case 'g':
                    case 'G':
                        istrstream >> parameters.gain;
                        break;
                    case 'f':
                    case 'F':
                        parameters.input_filename = new char[strlen(cmd) + 1];
                        istrstream >> parameters.input_filename;

```

```

                        break;
                    case 'm':
                    case 'M':
                        parameters.message = new char[strlen(cmd) + 1];
                        istrstream >> parameters.message;
                        break;
                    case 'z':
                    case 'Z':
                        istrstream >> parameters.gamma;
                        break;
                    default:
                        break;
                }
                // Lop off the last argument by replacing the dash with a NULL;
                *dash_ptr = '\0';
            } while (dash_ptr != NULL);

            //if (parameters.message == NULL)
            // {
            //     parameters.message = new char(strlen("Default message") + 1);
            //     strcpy(parameters.message, "Default message");
            // }

            // Clean up.
            delete [] commands;
        }

        SignerParams::~SignerParams(void)
        {
            if (parameters.input_filename != NULL)
                delete [] parameters.input_filename;

            //if (parameters.message != NULL)
            //     delete [] parameters.message;

            if (parameters.output_filename != NULL)
                delete [] parameters.output_filename;

            if (parameters.registry_name != NULL)
                delete [] parameters.registry_name;
        }

        // SignerParams::UpdateSignature()
        // Update the timestamp member variable within this object.
        // void SignerParams::UpdateSignature(void)
        // {
        //     // Set the timestamp indicating when we signed this puppy.
        //     CTime t = CTime::GetCurrentTime();
        //     parameters.sign_time = t;
        // }

        // .....
        // FILE: Params.h
        // .....
        // DESCRIPTION:
        // The Params classes are responsible for gathering and managing all
        // user input parameters. There are two classes defined here: 1) the
        // SignerParams class for the signer, and the ReaderParams class for the
        // reader.
        //
        // The SignerParams class also keeps track of internal parameters which
        // control or "tune" the operation of the signer, but which are not
        // accessible by the user.
        //
        // At present, this is a non-GUI version. All
        // user inputs enter from the command line. In the future, a GUI version
        // will be added which will present a dialog box to the user and gather
        // input parameters from a graphical interface. The command line version
        // will probably always exist for testing purposes and possibly batch
        // processing. Different constructors will be used to differentiate
        // between the GUI and cmd line versions.
        //
        // This header file should be included by any module which creates or
        // makes use of SignerParams and/or ReaderParams objects.

```



```

* CREATION DATE: August 15, 1995
*
* Copyright (c) 1995 Digimarc Incorporated, all rights reserved.
* .....
* #endif PARAMS_H
* #define PARAMS_H
*
* // #include "digimarc.h"
* #include <time.h>
* #include "stdafx.h"
*
* ///////////////////////////////////////////////////
* // SIGNER PARAMETERS STRUCTURES AND CLASSES
* ///////////////////////////////////////////////////
*
* typedef unsigned User_key_t;
*
* // Define a structure which will contain the various signer parameters.
* // The Signer Params class will contain a private copy of this structure.
* typedef struct
* {
*     // User inputs...
*     char *input_filename;
*
*     // User provides some combination of following to uniquely locate
*     // the registry entry for the signing event...
*     User_key_t user_key;
*     time_t date_of_signing;
*
*     char *registry_name; // optional
*
*     // "Super user" inputs, useful for testing and tuning, go here.
*
*     // Non user inputs will go here...
* } reader_param_struct;
*
* class ReaderParams
* {
* public:
*     ReaderParams(int argc, char *argv[]); // Constructor for non-gui (cmd line) version
*
*     // Create an accessor which returns a ptr to a const copy of the parameters structure.
*     // An alternative is to write accessors for each individual parameter.
*     const reader_param_struct *getParams(void) const;
*
*     // Private member functions and data structures
* private:
*     reader_param_struct parameters; // structure containing the user parameters.
*
*     // Function which warns user if parameters are not all present or look incorrect.
*     // It will also throw an exception if things are not right.
*     checkParams(void);
*
* };
*
* #endif // PARAMS_H
*
* ///////////////////////////////////////////////////
* // PARAMS.DLG.CPP
* ///////////////////////////////////////////////////
*
* // paramsdlg.cpp : implementation file
*
* #include "stdafx.h"
* #include "signer.h"
* #include "paramsdlg.h"
*
* #ifdef _DEBUG
* #undef THIS_FILE
* static char BASED_CODE THIS_FILE[] = __FILE__;
* #endif
*
* ///////////////////////////////////////////////////
* // ParamsDlg dialog
*
* ParamsDlg::ParamsDlg(CWnd* pParent /*=NULL*/)
* {
*     // CDialog(ParamsDlg::IDD, pParent)
*
*     m_message = "";
*     m_gain_from_edit_box = (float) 0.0;
*     m_key = 0;
*     m_bump_size = 0;
*     m_gamma_lut_scale = 0.0f;
*     // JAFX_DATA_INIT
*
* }
*
* void ParamsDlg::DoDataExchange(CDataExchange* pDX)
* {
*     CDialog::DoDataExchange(pDX);
*     // JAFX_DATA_MAP(ParamsDlg)
*     DDX_Text(pDX, IDC_MESSAGE, m_message);
*     DDX_MaxChars(pDX, m_message, 256);
*
* }

```

```

DDX_Text(pDX, IDC_EDIT_GAIN, m_gain_from_edit_box);
DDV_MinMaxFloat(pDX, m_gain_from_edit_box, 1.e-003f, 1.e+006f);
DDX_Text(pDX, IDC_EDIT_KEY, m_key);
DDV_MinMaxInt(pDX, m_key, 1, 256);
DDX_Text(pDX, IDC_EDIT_SIZE, m_bump_size);
DDV_MinMaxInt(pDX, m_bump_size, 1, 256);
DDX_Text(pDX, IDC_EDIT_SCALE, m_detail_lut_scale);
DDV_MinMaxFloat(pDX, m_detail_lut_scale, 1.e-003f, 1.e+006f);
//}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(ParmDlg, CDialog)
//{AFX_MSG_MAP(ParmDlg)
ON_COMMAND(ID_SETTINGS_SIGNER, OnSettingsSigner)
//}AFX_MSG_MAP
END_MESSAGE_MAP()

// ParmDlg message handlers
void ParmDlg::OnOK()
{
    CDialog::OnOK();
}

void ParmDlg::OnSettingsSigner()
{
    // TODO: Add your command handler code here
}

// ParmDlg.h : header file
//
#include "stdafx.h"
//
// ParmDlg dialog
//

class ParmDlg : public CDialog
{
public:
    ParmDlg(CWnd* pParent = NULL); // standard constructor

// Dialog Data
//{{AFX_DATA(ParmDlg)
enum { IDD = IDD_PARAMS_DIALOG };
CString m_message;
float m_gain_from_edit_box;
UINT m_key;
int m_bump_size;
float m_detail_lut_scale;
//}AFX_DATA

// Implementation
protected:
virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support

// Generated message map functions
//{{AFX_MSG(ParmDlg)
virtual void OnOK();
afx_msg void OnSettingsSigner();
//}AFX_MSG
DECLARE_MESSAGE_MAP()
};

```

## RAWIMAGE.H

```

//*****
// FILE: RawImage.h
//
// DESCRIPTION:
// RawImage objects are used to convert images from popular formats
// to the raw image format used internally by the Digimarc system.
// Typically, the RawImage constructor is given an input file as an
// argument, and the constructor is responsible for reading the file
// and performing the necessary operations to convert it into the raw
// format.
//
// RawImage objects also are able to perform the inverse conversion,
// creating image files in various standard formats from the internal
// raw representation.
//
// The initial implementation will only except TIFF files as inputs.
//

```

```

* and will make use of the public domain software LibTIFF in order*
* to read and write TIFF files.
*
* This header file should be included by any module which creates or*
* makes use of RawImage objects.
*
* CREATION DATE: August 15, 1995
*
* Copyright (c) 1995 Digimarc Incorporated, all rights reserved.*
\*****
#define RAWIMAGE_H
#include "digimarc.h"
#include "Params.h"

// Since the exact internal representation may change, use a typedef.
// This will allow a single change to modify all references to the
// raw image data format.
// Also note that in the future we will need several raw image representations.
typedef long * Raw_Data;

class RawImage
{
public:
    RawImage(SignerParams *params);

    // Member function which gives caller access to the raw image and its attributes.
    const int GetXdim(void);
    const int GetYdim(void);

    // This accessor returns a const pointer to a read-only image.
    const Raw_Data GetImage(void) const;

    // This accessor returns a const pointer to a writable image.
    Raw_Data * GetWritableImage(void) const;

    // Member function used to convert the raw image to an output TIFF file.
    WriteTIFF(char *filename);

// Private data. Users of rawImage objects get at these through accessors only.
private:
    int xdim; // X dimension of image
    int ydim; // Y dimension of image
    Raw_Data image; // Ptr to array of image data
};

#endif // RAWIMAGE_H

//*****
// FILE: Read.cpp
//
// DESCRIPTION:
// Core recognition functions of the Digimarc technology
// Created August 1995
//
// This particular code uses "raster" based processing as opposed to 2D based
//
// Copyright (c) 1996 Digimarc Corporation, all rights reserved.
//
// Include "read.h"
// Include "sign.h"
// Include "eff.h"
// Include "stdafx.h"
// Include "math.h"
//
// Constants */
const float epsilon = (float) 0.000001;

// read_8bit_single_channel_or_color()
//
// Used to read (or "recognize") the embedded digimarc signature in
// either a gray-scale or color image. Set number_channels to 1 for
// gray-scale, 3 for color.
//
// inc read_8bit_single_channel_or_color{
// unsigned char *data; // Input data to be recognized */
// long original_xdim, // It's x dimension */

```

```

long original_ydim,
long x_offset,
long y_offset,
long x_extent,
long y_extent,
int message_length,
unsigned char *key,
long key_length,
/* unused */
char *key_lut,
float *luminance_lut,
float *detail_lut,
unsigned char *thumbnail,
unsigned char *original_data,
/* if available, use pointer, otherwise NULL */
const unsigned char *referenceBitArray, // bit array ptr: either the known message or estimate.
float *metric, // we will compute a return a crude metric indicating confidence.
float *range,
unsigned char *message,
int number_channels,
int reading_mode,
int bumps
){
    int status = 1;

    if(reading_mode == 0){
        read_8bit_single_channel_OLD_plus_color(
            data, original_xdim, original_ydim, x_offset, y_offset,
            x_extent, y_extent, message_length, key, key_length, key_lut,
            luminance_lut, detail_lut, thumbnail, original_data, referenceBitArray,
            metric, range, message, number_channels, bumps;
        )
    }
    else if(reading_mode == 1){
        read_super(
            data, original_xdim, original_ydim, x_offset, y_offset,
            x_extent, y_extent, message_length, key, key_length, key_lut,
            luminance_lut, detail_lut, thumbnail, original_data, referenceBitArray,
            metric, range, message, number_channels, bumps;
        )
    }
    return(status);
}

// read_8bit_single_channel_OLD_plus_color()
// void read_8bit_single_channel_OLD_plus_color(
//     unsigned char *data, // input data to be recognized */
//     long original_xdim, // it's x dimension */
//     long original_ydim, // it's y dimension */
//     long x_offset, // x offset of segment */
//     long y_offset, // y offset of segment */
//     long x_extent, // x extent of segment */
//     long y_extent, // y extent of segment */
//     int message_length, // length of message in BITS, also length of message string */
//     unsigned char *key, // original 8 bit random key */
//     long key_length, // key_length often equal to data_length but not always */
//     char *key_lut, // look up table mapping key value */
//     float *luminance_lut, // look up table mapping the signature level to luminance */
//     float *detail_lut, // look up table mapping the signature level to local detail */
//     unsigned char *thumbnail, // if available, use pointer, otherwise NULL */
//     const unsigned char *referenceBitArray, // bit array ptr: either the known message or estimate.
//     float *metric, // we will compute a return a crude metric indicating confidence.
//     float *range,
//     unsigned char *message,
//     int number_channels,
//     int bumps
// ){
//     unsigned char *pkey, *pdata;
//     long i, line, bit;
//     int temp, status=1;
//     float *key_value = new float[x_extent];
//     float *data_float = new float[x_extent];
//     float *orig_float = new float[x_extent];
//     float *bit_total = new float[message_length];
//     //float *bit_mag = new float[message_length];
//     float *pkey_value, *pdata_float;

```

```

float filter_cf = (float)0.5; // kludge for now
double maxdiff = 40.0; // kludge for now

int key_length = 1*(original_xdim-1)/bumps;
for(i=0; i<message_length; i++){
    bit_total[i] = (float) 0.0;
    //bit_mag[i] = (float) 0.0;
}
pdata = data;
for(line=y_offset; line<(y_offset+y_extent); line++){
    /* FIRST: if either the original image or a thumbnail of the original is available,
    then use either a simple or "advanced" dot product to remove it; "advanced" refers
    to the idea that you may wish to adjust the gamma or higher order stuff */
    float it(pdata, data_float, x_extent, number_channels);
    //derivative threshold(data_float, x_extent, number_channels, maxdiff, filter_cf);
    //remove_mean(data_float, x_extent);
    /* load key values */
    int key_offset = (line/bumps)*key_xlength;
    pkey = &key[key_offset + x_offset/bumps];
    pkey_value = key_value;
    if(bumps>1){
        for(i=x_offset; i<(x_offset+x_extent); i++){
            *pkey_value++ = (float){ (int)key_lut[ (int)*pkey ] };
            if( !(i%1)*bumps ) pkey++;
        }
    }
    else {
        for(i=x_offset; i<(x_offset+x_extent); i++){
            *pkey_value++ = (float){ (int)key_lut[ (int)*pkey++ ] };
        }
    }
    pdata++=(number_channels*x_extent);
}

/* now step through processed patch and perform simple or "advanced" correlation
detection, keeping the resultant detection values in the accumulators for each bit of the
message_length
bits */
pdata_float = data_float;
pkey_value = key_value;
float running_average = (float) 0.0;
float ftemp;
for(i = 0; i < MOV_AV_KERNEL; i++){
    running_average += *pdata_float++;
}
float mov_av = (float)MOV_AV_KERNEL;
running_average /= mov_av;
pdata_float = data_float;
temp = MOV_AV_KERNEL/2;
int temp1 = temp-1;
if(bumps>1){
    for(i = x_offset; i < (x_offset + x_extent); i++){
        if(i <= (x_offset + temp) || i >= (x_offset + x_extent - temp) ){
            ftemp = *(pdata_float + temp) - *(pdata_float - temp1) / mov_av;
            running_average += ftemp;
        }
        bit = (key_offset + i)/bumps;
        temp = *(pdata_float++) - running_average;
        //bit_mag[bit] += (*pkey_value - *pkey_value);
        bit_total[bit] += (ftemp * (*pkey_value));
    }
}
else {
    for(i = x_offset; i < (x_offset + x_extent); i++){
        if(i <= (x_offset + temp) || i >= (x_offset + x_extent - temp) );
        else
            ftemp = *(pdata_float + temp) - *(pdata_float - temp1) / (float)
            running_average += ftemp;
        float mov_av = (float)MOV_AV_KERNEL;
        running_average /= ftemp;
        bit = (key_offset + i) / message_length;
        //bit_mag[bit] += (*pkey_value - *pkey_value);
        bit_total[bit] += (ftemp * (*pkey_value));
    }
}
// time optimized version of above earlier code
int key_tot = key_offset + x_offset;
for(i=x_offset; i<(x_offset+temp); i++){

```

```

        bit = key_foo++ % message_length;
        bit_total[bit] += ( (pdata_float++) - running_average ) * (pkey_value++);
    }
    int temp2 = x_offset + x_extent - temp;
    float *pdata_float2 = data_float;
    float *pdata_float1 = pdata_float(temp);
    for(i=x_offset-temp; i<temp; i++){
        running_average += ( (pdata_float1++) - (pdata_float2++) ) / mov_av;
        bit = key_foo++ % message_length;
        bit_total[bit] += ( (pdata_float++) - running_average ) * (pkey_value++);
    }
    for(i=0; i<temp; i++){
        bit = key_foo++ % message_length;
        bit_total[bit] += ( (pdata_float++) - running_average ) * (pkey_value++);
    }
}

/* fill the message string based on bit_totals */
for(i=0; i<message_length; i++)
{
    if(bit_total[i]>0.0)
    {
        message[i]=1;
    }
    else
    {
        message[i]=0;
    }
}

/*
for (i = 0; i < message_length; i++)
{
    // Before normalizing by the magnitudes, be sure we aren't
    // dividing by zero (this happens for an image w/ zero energy.
    if (bit_mag[i] == (float)0.0)
        bit_mag[i] = epsilon;
    bit_total[i] /= (float) sqrt( (double) bit_mag[i] );
}
*/

// Compute the "crude metric", an estimate of rms spread of the
// bit level detector's results. The referenceBitArray is either
// the known message (if it was available to caller) or the
// newly computed estimate of the message.
*metric = get_crude_metric(referenceBitArray, bit_total, range, message_length);

delete () data_float;
delete () orig_float;
delete () bit_total;
delete () key_value;
delete () bit_mag;

return;
}

// float_it()
// void float_it(unsigned char *data, float *data_float,
// int number_channels, int number_channels,
// int number_channels)
// {
//     unsigned char *pdata;
//     long i;
//     float *pdata;
//     pdata = data;
//     if(number_channels == 1){
//         for (i = 0; i < x_extent; i++){
//             *pdata++ = (float) *pdata++;
//         }
//     }
//     else if (number_channels == 3) {
//         for (i = 0; i < x_extent; i++){
//             *pdata = (float) *pdata++;
//             *pdata = (float) *pdata++;
//             *pdata = (float) *pdata++;
//         }
//     }
// }

// int derivative_threshold(float *data, long length, int number_channels, double maxdiff, float
// filter_cf)
// {
//     long i;
//     int status = 1;
//     float *pdata_llast, last;
//     double diff;
//     float replacement = (float)0.0;
//     if(number_channels == 3) maxdiff *= 3.0;
//     last = llast = data[0];
//     pdata = &data[1];

```

```

for(i=0; i<length; i++){
    diff = (double)*pdata - last;
    last = *pdata;
    if (fabs(diff) > maxdiff){
        if (diff > 0) diff = replacement;
        else diff = -replacement;
    }
    *pdata = last + (float)diff;
    last = *pdata++;
}

return(status);
}

void read_super(
    unsigned char *data,
    long original_xdim,
    long original_ydim,
    long x_offset,
    long y_offset,
    long x_extent,
    long y_extent,
    int message_length,
    unsigned char *key,
    long key_length,
    /* unused */ length,
    /* input data to be recognized */
    /* it's x dimension */
    /* it's y dimension */
    /* x offset of segment */
    /* y offset of segment */
    /* x extent of segment */
    /* y extent of segment */
    /* length of message in bits, also length of message string */
    /* original 8 bit random key */
    /* key_length often equal to data_length but not always */

    /* look up table mapping key value */
    /* look up table mapping the signature level to luminance */
    /* look up table mapping the signature level to luminance */
    /* if available, use pointer, otherwise NULL */
    /* if available, use pointer, otherwise NULL */

    unsigned char *thumbnail,
    unsigned char *original_data,
    const unsigned char *reference_bit_array, // bit array ptr: either the known message or estimate.
    float *range,
    float *metric,
    unsigned char *message,
    int number_channels,
    int bumps
){
    unsigned char *pkey, *pdata;
    long i, line, bit;
    int status=1, bits, fftdim, j, highest;
    float *bit_total = new float(message_length);
    float *bit_mag = new float(message_length);
    float *key_value = new float(x_extent)*pkey_value;
    int key_xlength = 1*(original_xdim-1)/bumps;

    for(i=0; i<message_length; i++){
        bit_total[i] = (float) 0.0;
        bit_mag[i] = (float) 0.0;
    }

    // find power of 2 higher than highest dimension
    if(x_extent > y_extent) highest = x_extent;
    else highest = y_extent;
    bits = 1 + (int)(log( (double)highest - 0.5 ) / log(2.0) );
    fftdim = (int)pow(2.0, (double)bits + 0.00000001);

    // create array
    float *image = new float(fftdim*(fftdim+2));
    float *w = new float(fftdim);
    float *wi = new float(fftdim);
    float *pimage;
    pimage = image;
    for(i=0; i<(fftdim*(fftdim+2)); i++){
        *pimage++ = (float)0.0;
    }

    // convert either a B&W image or a color image to a single floating point luminance image
    float total;
    if(number_channels == 1){
        pdata = data;
        for(i=0; i<y_extent; i++){
            pimage = image(i*fftdim);
            for(j=0; j<x_extent; j++){
                *pimage = (float)*pdata++;
                total += *pimage++;
            }
        }
    }
    else if(number_channels == 3){
        pdata = data;
        for(i=0; i<y_extent; i++){
            pimage = image(i*fftdim);
            for(j=0; j<x_extent; j++){
                *pimage = (float)*pdata++;
                *pimage++ = (float)*pdata++;
                *pimage++ = (float)*pdata++;
            }
        }
    }

    // weird derivative threshold
    int choo=0;
    if(choo){
        // remove dc
        total /= ((float)y_extent * (float)x_extent);
        for(i=0; i<y_extent; i++){
            pimage = image(i*fftdim);
            for(j=0; j<x_extent; j++){
                *pimage++ -= total;
            }
        }
    }

    float *detail_vector;
    float *detail_vector = new float(x_extent);
    int start = 5;
    int stop = 500;
    float scale = (float)0.5;
    for(i=0; i<y_extent; i++){
        get_read_detail_vector(detail_vector, data, x_extent, i, y_extent, number_channels, start, stop, scale,
            image, fftdim);
        detail_vector = detail_vector;
        pimage = image(i*fftdim);
        for(j=0; j<x_extent; j++){
            *pimage++ += *pdetail_vector++;
        }
        delete [] detail_vector;
    }

    //float filter_cf = (float)0.5; // kludge for now
    //double maxdiff = 40.0; // kludge for now
    //for(line=0; line<y_extent; line++){
    //    derivative_threshold(image(line*fftdim), x_extent, 1, maxdiff, filter_cf);
    //}

    // easy does the window ??
    // for now, multiply the last four values near the edges by a linear ramp to zero, simply
    // to avoid total edge weirdnesses
    int window_it=0;
    if(window_it){
        if(x_extent > 10 && y_extent > 10){
            float mult[4];
            mult[0] = (float)0.2, mult[1] = (float)0.4, mult[2] = (float)0.6, mult[3] = (float)0.8;
            pmult = mult;
            for(i=1; i<5; i++){
                pimage = image((i-1)*fftdim);
                for(j=0; j<x_extent; j++){
                    *pimage++ *= *pmult++;
                }
            }
            pmult = mult;
            for(i=1; i<5; i++){
                pimage = image((y_extent - i)*fftdim);
                for(j=0; j<x_extent; j++){
                    *pimage++ *= *pmult++;
                }
            }
        }
    }

    // fft arrays
    realfft2d_in_place(image, bits, 0, w, wi);

    // filter them
    // phase difference only to start
    // calculate phase differences and reload them into real and imaginary1
    float mag1, preall, *pimaginary1;
    // double power = 0.8;
    preall = image;
    pimaginary1 = image(fftdim);
    for(i=0; i<(fftdim/2); i++){
        for(j=0; j<fftdim; j++){
            mag1 = (float)fabs( (double)*preall + (float)fabs( (double)*pimaginary1 );
            if(mag1 == (float)0.0){
                *preall++ = (float)0.0;
            }
        }
    }

```

```

}
else {
    mag1 = (float)pow((double)mag1,power);
    preali1 += mag1;
    pimaginary1 += mag1;
}
}
preali1 -= fftdim;
pimaginary1 -= fftdim;
}

// remove low and/or high frequencies
// the DC should reside at row one, fftdim/2
int row = 0;
if(mod){
    int low = 1;
    int account_low = 2-1;
    pimage = image[(fttdim/2) - low + 1];
    for(j=0; j<account; j++){
        pimage += (float)0.0;
        pimage -= (fttdim - account);
    }
}

// inverse fft
realftid_in_place(image,bits,1,wr,w1);
for(line=y_offset; line<(y_offset+y_extent); line++){
    /* load key values */
    pkey = key[(line/bumps) * key_length + x_offset/bumps];
    for(i=x_offset; i<(x_offset+x_extent); i++){
        key_value[i-x_offset] = (float)((int)key_lut[(int)pkey]);
        if((i+1)%bumps)pkey++;
    }

    /* now step through processed patch and perform simple or "advanced" correlation detection.
    keeping the resultant detection values in the accumulators for each bit of the
    message_length
    bits =
    pimage = image[(line-y_offset)*fttdim];
    pkey_value = key_value;
    for(i=x_offset; i<(x_offset+x_extent); i++){
        bit = ((line/bumps)*key_length + i/bumps) % message_length;
        bit_mag[bit] += (*pkey_value * pkey_value);
        bit_total[bit] += (*pimage++) * (*pkey_value++);
    }
}

/* fill the message string based on bit_totals */
for(i=0; i<message_length; i++){
    if(bit_total[i]>0.0)
        message[i]=1;
    else
        message[i]=0;
}

for (i = 0; i < message_length; i++)
    // Before normalizing by the magnitudes, be sure we aren't
    // dividing by zero (this happens for an image w/ zero energy.
    if (bit_mag[i] == (float)0.0)
        bit_mag[i] = epsilon;
    bit_total[i] /= (float) sqrt( (double) bit_mag[i] );

    // Compute the "crude metric", an estimate of rms spread of the
    // bit level detector's results. The referenceArray is either
    // the known message (if it was available to caller) or the
    // newly computed estimate of the message.
    *metric = get_crude_metric(referenceBitArray, bit_total, range, message_length);

    delete [] bit_total;
    delete [] bit_mag;
    delete [] key_value;
    delete [] image;
    delete [] wr;
    delete [] w1;
}

return;
}

// Header file for the Reader core algorithm functions.
// READ.B

```

-39-





```

////////////////////////////////////
// get_detail_vector()
//
int get_detail_vector(
    float *detail_lut,
    unsigned char *data,
    int xdim,
    int row,
    int total_rows,
    float *detail_lut,
    int number_channels
)
{
    unsigned char *pdata, *p1, *p2;
    int base, temp, i;
    float *pdetail_vector, *pdetail_vector_detail_vector;

    // this function creates a "scaling" vector for the current scan line,
    // based on a crude metric of "local detail"
    if (number_channels == 1) {
        pdata = data;
        if (row == 0) p1 = data;
        else p1 = data - xdim;
        if (row == (total_rows-1)) p2 = data;
        else p2 = data + xdim;
        // perform first and last elements outside loop so that an internal if statement is avoided
        base = (int)*(pdata+1);
        temp = abs(base - (int)*(p1+1));
        temp += abs(base - (int)*(p2+1));
        temp += abs(base - (int)*(p2+1));
        temp += 2*abs(base - (int)*pdata);
        for (i=1; i<xdim-1; i++) {
            base = (int)*(pdata+i);
            temp = abs(base - (int)*(p1+i));
            temp += abs(base - (int)*(p2+i));
            temp += abs(base - (int)*pdata);
            temp += abs(base - (int)*(pdata-2));
            *(pdetail_vector++) = detail_lut[temp];
        }
        base = (int)*pdata;
        temp = abs(base - (int)*p1);
        temp += abs(base - (int)*p2);
        temp += 2*abs(base - (int)*(pdata-1));
        *pdetail_vector = detail_lut[temp]; // make sure it goes up to 1024 elements
    }
    else if (number_channels == 3)
    {
        // use the green channel only just for speed's sake
        pdata = data+1;
        if (row == 0) p1 = data+1;
        else p1 = data+1 - 3*xdim;
        if (row == (total_rows-1)) p2 = data+1;
        else p2 = data+1 + 3*xdim;
        // perform first and last elements outside loop so that an internal if statement is avoided
        base = (int)*pdata;
        temp = abs(base - (int)*p1);
        temp += abs(base - (int)*p2);
        temp += 2*abs(base - (int)*pdata);
        *pdetail_vector++ = detail_lut[temp]; // make sure it goes up to 1024 elements
        for (i=1; i<(xdim-1); i++)
        {
            base = (int)*pdata;
            temp = abs(base - (int)*p1);
            temp += abs(base - (int)*p2);
            temp += 2*abs(base - (int)*pdata);
            *pdetail_vector++ = detail_lut[temp];
        }
        base = (int)*pdata;
        temp = abs(base - (int)*p1);
        temp += abs(base - (int)*p2);
        temp += 2*abs(base - (int)*(pdata-3));
        *pdetail_vector = detail_lut[temp]; // make sure it goes up to 1024 elements
    }
    return(1);
}

////////////////////////////////////
// load_detail_lut()
//
// This function loads the scaling factor based on local detail
int load_detail_lut( float *detail_lut, float scale) // explicitly written for 8 bit
{
    int i, status=1;
    float length=(float) (DETAIL_STOP-DETAIL_START);
    scale /= (float)100.0;

```

```

scale*=DETAIL_NORMALIZER;
for(i=0; i<DETAIL_START; i++) detail_lut[i]=(float)1.0;
// for(i=DETAIL_START; i<DETAIL_STOP; i++)
// {
//     detail_lut[i] = (float)1.0 + scale*((float)(i-DETAIL_START)/length);
// }
for(i=DETAIL_STOP; i<DETAIL_TOTAL; i++) detail_lut[i]=detail_lut[DETAIL_STOP-1];

return(status);
}

////////////////////////////////////
// sign_8bit_single_channel_or_color()
//
// written for the march 1996 bump incarnation
//
// int sign_8bit_single_channel_or_color(
//     unsigned char *data,
//     long data_length,
//     long xdim,
//     long ydim,
//     unsigned char *message,
//     int message_length,
//     unsigned char *key,
//     long key_length,
//     *unused,
//     char *key_lut,
//     float *luminance_lut,
//     float *detail_lut,
//     int signing_mode,
//     unsigned char *data_out,
//     int number_channels,
//     color_images
// )
// {
//     unsigned char *pdata;
//     unsigned char *p_out;
//     unsigned char *pkey;
//     unsigned char *pmessage;
//     long i;
//     int j, k;
//     int lum_change, status=1;
//     float ftemp, delta;
//     float *pdetail_vector = new float[xdim];
//     float *pdetail_vector, local_gain;
//     int key_xlength;
//     key_xlength = 1+(xdim-1)/bumps;
//     if (number_channels == 1) {
//         pdata = data;
//         p_out = data_out;
//         for(i=0; i<ydim; i++) {
//             // load local detail values for this row
//             get_detail_vector(detail_vector, pdata, xdim, i, ydim, detail_lut, number_channels);
//             pdetail_vector = detail_vector;
//             pkey=key[(i/bumps)*key_xlength];
//             pmessage = *message[(i/bumps)*key_xlength]*message_length;
//             for(j=0; j<xdim; j++) {
//                 lum_change = key_lut[(int)*pkey];
//                 if (lum_change == -1)
//                     *(p_out++) = *(pdata++);
//                 else
//                     local_gain = *(pdetail_vector++) * luminance_lut[*pdata];
//                 if (abs(lum_change) > 1) { // this is the anti-sparkles check
//                     if (local_gain > (float)3.5) {
//                         if (lum_change > 0) lum_change = 1;
//                         else lum_change = -1;
//                     }
//                 }
//                 delta = (float)lum_change * local_gain;
//                 if (i*(pmessage))
//                     delta = -delta; // invert current snowy image luminance value ... key
//                 ftemp = (float)*(pdata++) * delta;
//                 if (ftemp > (float)255.0) *(p_out++) = (unsigned char)255;
//                 else if (ftemp < (float)0.0) *(p_out++) = (unsigned char)0;
//                 else *(p_out++) = (unsigned char)(ftemp*(float)0.5);
//             }
//             if ((j+1)%bumps) == 0 {

```

```

key++;
if( (((bumps)*key_xlength+j/bumps)*message_length) == (message_length-1) ) /*
time to restart message */
{
    pmessage = message;
}
else pmessage++;
}
}
}
else if(number_channels == 3){
// data_length is assumed to be the number of pixels, not the number of data bytes
// RGB packing is assumed, in that order, 3 bytes in a row per pixel: R G B
if(signing_mode == STANDARD){
    pdata = data;
    p_out = data_out;
    for(i=0;i<ydim;i++){
        // load local detail values for this row
        get_detail_vector(detail_vector,pdata,xdim,i,ydim,detail_lut,number_channels);
        detail_vector = detail_vector;
        pkey=key/(i/bumps)*key_xlength;
        pmessage = pmessage+(((i/bumps)*key_xlength)*message_length);
        for(j=0;j<xdim;j++){
            lum_change = key_lut((int)*pkey);
            if(lum_change == 0){
                memcpy(p_out,pdata,3*sizeof(unsigned char));
                p_out++;
                p_out++;
                p_out++;
                pdetail_vector++;
            }
            else {
                local_gain = *(pdetail_vector++) * luminance_lut(*(pdata+1));
                if( abs(lum_change) > 1 ){ // this is the anti-sparkles check
                    if( local_gain > (float)3.5 ){
                        else lum_change = 1;
                    }
                    else lum_change = -1;
                }
            }
            delta = (float)lum_change * local_gain;
            if( !(*pmessage) )
                delta = -delta; /* invert current snowy image luminance value ... key */
            for(k=0;k<3;k++){
                ftemp = (float)*(pdata++) + delta;
                if(ftemp > (float)255.0)*(p_out++) = (unsigned char)255;
                else if(ftemp < (float)0.0)*(p_out++) = (unsigned char)0;
                else *(p_out++) = (unsigned char)(ftemp*(float)0.5);
            }
        }
        if( ((j+1)%bumps) == 0 ){
            key++;
            if( (((bumps)*key_xlength+j/bumps)*message_length) == (message_length-1) )
                /* time to restart message */
            {
                pmessage = message;
            }
            else pmessage++;
        }
    }
}
return(status);
}

//////////////////////////////////////
// FILE: SignDoc.cpp
//
// DESCRIPTION:
// Implementation file for the Document class of the Digimarc Signer.
// This defines the implementation of the document class
// for the Signer. Under the Microsoft Foundation Class (MFC) architecture,
// the Document/View model is the preferred method. This header file
// defines our additions to the generic Document class created by the
// Visual C++ wizards.
//
// Copyright (C) 1996 Digimarc Corporation, all rights reserved.
//
//////////////////////////////////////
#include "stdafx.h"
#include "signer.h"
#include "limits.h"
#include "signdoc.h"
#include "signview.h"
#include "cokey.h"
#include "image.h"
#include "sign.h"
#include "read.h"
#include "align.h"
#include "parmsdig.h"
#include "treaddig.h"
// For the Signer Parameters dialog object
// For the Reader Parameters dialog object
// These are the possible settings of the "signing_mode" argument
#define STANDARD 0
#define STRICT_LUMINANCE 1

```





```

void CDibDoc::Dump(CDumpContext& dc) const
{
    CDocument::Dump(dc);

    #endif // _DEBUG

    ////////////////////////////////////////////////////
    // DumpBitmapInfoHeader()
    ////////////////////////////////////////////////////
    // Diagnostic tool which dumps out some information about the DIB's
    // header. Only used for test/debug purposes.
    ////////////////////////////////////////////////////
    void CDibDoc::DumpBitmapInfoHeader() const
    {
        int i, cxDIB, cyDIB,
            num_pixels, num_colors;
        long
        LPSTR
        LPBITMAPINFOHEADER lpDIBHdr;
        LPBITMAPINFO
        lpBmi;

        HDB hOriginalDIB = GetOriginalHDB();
        if (hOriginalDIB == NULL)
            return;

        // Lock the DIB in memory
        lpDIB = (LPSTR)::GlobalLock((HGLOBAL) hOriginalDIB);

        // Get ptr to the dib header space.
        lpDIBHdr = (LPBITMAPINFOHEADER) lpDIB;

        // get pointer to BITMAPINFO (Win 3.0)
        lpBmi = (LPBITMAPINFO) lpDIB;
        RGBQUAD *bmiColors = lpBmi->bmiColors;

        cxDIB = (int)::DIBWidth(lpDIB); // X size of DIB
        cyDIB = (int)::DIBHeight(lpDIB); // Y size of DIB
        num_pixels = (long) cxDIB * cyDIB;
        num_colors = ::DIBNumColors(lpDIB);

        if (lpDIBHdr->biCompression != 0)
        {
            TRACE("Can't cope with compressed image (compression = %d)\n",
                  lpDIBHdr->biCompression);
            ::GlobalUnlock((HGLOBAL) m_hOriginalDIB);
            return;
        }

        TRACE("BITMAPINFOHEADER contents are:\n");
        TRACE("HeaderSize = %d, width = %d, height = %d, num_pixels = %d\n",
              lpDIBHdr->biSize, cxDIB, cyDIB, num_pixels);
        TRACE("planes = %d, bitsPerPixel = %d\n",
              lpDIBHdr->biPlanes, lpDIBHdr->biBitCount);
        TRACE("compressionMethod = %d\n", lpDIBHdr->biCompression);
        TRACE("SizeOfBitmap = %d\n", lpDIBHdr->biSizeImage);
        TRACE("num_colors = %d\n", num_colors);
        TRACE("compression = %d, verticalResolution = %d\n",
              lpDIBHdr->biCompression, lpDIBHdr->biVerticalResolution);
        TRACE("numColors = %d, numSignColors = %d\n",
              lpDIBHdr->biClrUsed, lpDIBHdr->biClrImportant);

        // Dump the palette. This is only for severe debugging situations.
        TRACE("\nThe contents of the palette:\n");
        for (i = 0; i < num_colors; i++)
        {
            TRACE("%d %2x %2x %2x\n", i,
                  (int) bmiColors->rgbRed,
                  (int) bmiColors->rgbGreen,
                  (int) bmiColors->rgbBlue);
            bmiColors++;
        }

        // We are now all done w/ the Original DIB. Unlock it.
        ::GlobalUnlock((HGLOBAL) hOriginalDIB);
    }

    ////////////////////////////////////////////////////
    // Member function which
    // builds a snow image in place.
    ////////////////////////////////////////////////////
    //
    // typed char *HPSTR; // huge pointer to a string NOW OBSOLETE

```

```

    TRACE("At this time, only build snow image for 8 or 24 bit images\n");
    ::GlobalUnlock((HGLOBAL) m_hSnowyDIB);
    return;
}

if (m_BitsPerPixel == 8 || m_BitsPerPixel == 24)
{
    CoxKey coxkey(m_pParams->GetKey(), (BITMAPINFO *) lpSnowyDIBHdr,
        hpsSnowyDIBBits);
}

::GlobalUnlock((HGLOBAL) m_hSnowyDIB);
}

// Sign()
// This is the function which calls upon the core signing algorithms.
// WARNING: CURRENTLY THIS FUNCTION ASSUMES THAT WE ALWAYS ARE SIGNING
// THE "ORIGINAL IMAGE" DIB. THIS MAY BE A BUG.
// First shot at a function which calls the signer core algorithms
void CDbDoc::Sign(void)
{
    long num_pixels, num_colors;
    image_byte;
    DWORD src_data, dest_data; // Huge ptrs for copying the image.
    HPSTR tmp;
    float num_channels;
    int HDIB hOriginalDIB = GetOriginalHDIB();
    if (hOriginalDIB == NULL)
        return;

    // Create space for the signed image DIB
    m_hSignedDIB = (HDIB) ::GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT, m_dwTotalDIBSize);
    if (m_hSignedDIB == 0)
    {
        MessageBox(NULL,
            "Insufficient memory is available for the signed image",
            "Digital Signer Warning",
            MB_ICONINFORMATION | MB_OK);
        return;
    }

    // Create image objects for the images. Note that this locks them in memory.
    Image snowyImage(m_hSnowyDIB);
    Image unsignedImage(m_hOriginalDIB);

    // This is ugly, but I have to copy the DIB header stuff into the signed DIB
    // before I can create the signedImage object.
    dest_data = (char *) ::GlobalLock((HGLOBAL) m_hSignedDIB);

    // We want to copy the BITMAPINFO structure from the unsigned to the signed DIB
    src_data = unsignedImage.GetUpDIB();

    // Copy the BITMAPINFOHEADER and palette to the signed DIB space, byte by byte.
    for (image_byte = 0; image_byte < unsignedImage.GetSizeofHeader(); image_byte++)
    {
        *dest_data++ = *src_data++;
    }

    ::GlobalUnlock((HGLOBAL) m_hSignedDIB);

    // Now create the signedImage object, which will lock the DIB in memory again.
    Image signedImage(m_hSignedDIB);

    // For each, create a "byte-wise" packed data array from the DIB 4-byte packing
    snowyImage.MakePackedData(FORCE_TO_1_CHANNEL); // snow image always 1 chan
    unsignedImage.MakePackedData();
    signedImage.MakePackedData();

    num_pixels = (long) unsignedImage.GetXDIM() * unsignedImage.GetYDIM();
    num_colors = unsignedImage.GetNumColors();

    if (m_BitsPerPixel != 8 && m_BitsPerPixel != 24)
    {
        TRACE("At this time, only sign 8 and 24 bit images\n");
        return;
    }

    // Create and load the luminance scaling look up table.
    float *luminance_lut = new float[256];
    ::load_luminance_lut(luminance_lut, m_pParams->GetGamma());

    // Create a packed msg (will be a user input in future).
    if (m_pPackedMsg != NULL)
        delete m_pPackedMsg;
    m_pPackedMsg = new PackedMsg( (const char *) m_pParams->GetMessage());

    // Set up some arguments and call the core signer.
    int x_dim = unsignedImage.GetXDIM();
    int y_dim = unsignedImage.GetYDIM();

    if (unsignedImage.GetBitsPerPixel() == 8)
    else if (unsignedImage.GetBitsPerPixel() == 24)
        num_channels = 3;

    // const float lut_scale = (float)1.0; // Later this will be user controlled.
    float detail_lut = new float[DETAIL_TOTAL];
    ::load_detail_lut(detail_lut, m_pParams->GetLutScale());

    // sign_8bit_single_channel_or_color(unsignedImage.GetPackedData(),
    // data_length,
    // x_dim,
    // y_dim,
    m_pPackedMsg->getMsgBitArray(),
    m_pPackedMsg->getMsgBitArrayLength(),
    snowyImage.GetPackedData(),
    data_length,
    key_lut,
    luminance_lut,
    detail_lut,
    STANDARD,
    signedImage.GetPackedData(),
    num_channels,
    m_pParams->GetBumpSize());

    delete [] detail_lut;

    // Set the timestamp indicating when we signed this puppy.
    m_pParams->UpdateSignature();

    delete [] luminance_lut;
    delete [] key_lut;

    // Now unpack the data in the image object, back into the standard DIB format
    signedImage.UnpackData();
}

// Read()
// The read function is the interface to the core recognition algorithm.
// It sets up the necessary data structures needed by the core routine
// and makes the call.
// void CDbDoc::Read(HDIB hSignedDIB, BOOL use_super_reader)
{
    long num_pixels, num_colors;
    int num_channels;
    int reading_mode;

    // Create image objects for the images. Note that this locks them in memory.
    Image snowyImage(m_hSnowyDIB);
    Image signedImage(hSignedDIB);

    // Create a "byte-wise" packed data array from the DIB 4-byte packing
    signedImage.MakePackedData();
    snowyImage.MakePackedData(FORCE_TO_1_CHANNEL); // Snowy images always 1 ch.
    // unsignedImage.MakePackedData();

    num_pixels = (long) signedImage.GetXDIM() * signedImage.GetYDIM();
    num_colors = signedImage.GetNumColors();

    if (m_BitsPerPixel != 8 && m_BitsPerPixel != 24)
    {

```

```

TRACE("At this time, only recognize 8 and 24 bit images\n");
return;
}

// Create and load the luminance scaling look up table.
float *luminance_lut = new float[256];
::load_luminance_lut(luminance_lut, m_pParams->GetGamma());

// Create and load the key look up table.
char *key_lut = new char[256];
::load_key_lut(key_lut, m_pParams->GetGain());

// Create and load the detail look up table.
float *detail_lut = new float[DETAILED_TOTAL];
//const float lut_scale = (float)1.0; // Later this will be user controlled.
::load_detail_lut(detail_lut, m_pParams->GetLutScale());

// Determine which bit array to use for the reader's "crude metric"
// computation. If we have just signed this image, then use the
// true message bit array. Otherwise, we are trying to read
// without knowing the true message, and use the estimated
// message for computation of the metric.
unsigned char *referenceBitArray;
if (m_state == IMAGE_SIGNED || m_state == IMAGE_SIGNED_AND_VERIFIED ||
    m_state == IMAGE_SIGNED_AND_SAVED)
    referenceBitArray = m_pPackedMsg->getMsgBitArray();
else
    referenceBitArray = m_pPackedMsg->getReaderBitArray();

long data_length = signedImage.GetXDim() * signedImage.GetYDim();
long x_offset = 0;
long y_offset = 0;
int x_dim = signedImage.GetXDim();
int y_dim = signedImage.GetYDim();

if (signedImage.GetBitsPerPixel() == 8)
    num_channels = 1;
else if (signedImage.GetBitsPerPixel() == 24)
    num_channels = 3;

// See if we should use the super reader.
if (use_super_reader)
    reading_mode = 1;
else
    reading_mode = 0;

// Call the core recognizer
::read_8bit_single_channel_or_color(
    x_dim,
    y_dim,
    x_offset,
    y_offset,
    x_dim,
    y_dim,
    // segment is full image.
    m_pPackedMsg->getMsgBitArrayLength(),
    snowImage.GetPackedData(),
    data_length,
    key_lut,
    luminance_lut,
    detail_lut,
    NULL,
    // No thumbnail at this time
    //unsignedImage.GetPackedData(),
    NULL,
    // Don't pass original data now
    (const unsigned char *) referenceBitArray,
    km_crude_metric,
    km_range,
    m_pPackedMsg->getReaderBitArray(),
    num_channels,
    reading_mode,
    m_pParams->GetBumpSize());

// Convert the recovered message bits back to an ASCII string.
m_pPackedMsg->bitsToString();

TRACE ("The recognizer detected the following string: %s\n",
    m_pPackedMsg->getRecoveredAsciiMsg());

delete [] luminance_lut;
delete [] key_lut;
delete [] detail_lut;
}

// CDibDoc commands

```

```

// OnSettingsDialog()
// This method is invoked when the user selects the Settings-->
// Signer Controls... menu item. It creates a signer parameters
// dialog object and presents it to the user as a modal dialog.
// If the user presses OK, we then gather the new parameter values
// and use them to sign the image. Finally, a new view and window
// are created to display the signed image, if no such view already
// exists.
//void CDibDoc::OnSettingsDialog()
//{
//    ParmDlg dlg;
//    CRect rect;
//    unsigned old_key;
//    BOOL new_user_key = FALSE;
//
//    // Check to see if we are in a legal state for signing.
//    if (m_state == NO_IMAGES)
//    {
//        MessageBox(NULL,
//            "An 8 or 24 bit image must be loaded before using the Signer.",
//            "Digimarc Signer Warning",
//            MB_ICONINFORMATION | MB_OK);
//        return;
//    }
//    int scroll_pos;
//
//    // Initialize the dialog data
//    dlg.m_message = m_pParams->GetMessage();
//    dlg.m_gain_from_edit_box = m_pParams->GetGain();
//    dlg.m_gamma = m_pParams->GetGamma();
//    dlg.m_key = m_pParams->GetKey();
//    old_key = m_pParams->GetKey();
//    dlg.m_bump_size = m_pParams->GetBumpSize();
//    dlg.m_detail_lut_scale = m_pParams->GetLutScale();
//
//    // Get the coordinates for the scroll bar object window.
//    dlg.m_gain.GetWindowRect(&rect);
//
//    // Try to "create" the scroll bar.
//    dlg.m_gain.Create(WS_CHILD, CRect(10, 50, 200, 20), &dlg, IDC_GAIN);
//
//    // Invoke the dialog box
//    if (dlg.DoModal() == IDOK)
//    {
//        // retrieve the dialog data
//        m_pParams->SetMessage(dlg.m_message);
//        if (dlg.m_key != old_key)
//        {
//            m_pParams->SetKey(dlg.m_key);
//            new_user_key = TRUE;
//        }
//        m_pParams->SetGain(dlg.m_gain_from_edit_box);
//        m_pParams->SetBumpSize(dlg.m_bump_size);
//        m_pParams->SetLutScale(dlg.m_detail_lut_scale);
//        m_pParams->SetGamma(dlg.m_gamma);
//        // gamma no longer user cntrl
//        scroll_pos = dlg.m_gain.GetScrollPos();
//        // TRACE("Scrollbar position: %d\n", scroll_pos);
//        // This is going to take awhile
//        BeginWaitCursor();
//
//        // NOTE: AT THIS POINT SHOULD DETERMINE WHAT IMAGE IS IN THE
//        // ACTIVE VIEW, AND IF IT CONTAINS A BITMAP SIGN THAT IMAGE.
//        // SEE OnSettingsReader(), which uses the correct logic.
//        // Then, call MakeSnow(hImageToSignDIB) and Sign(hImageToSignDIB)
//        // If the user seed has changed, or if we haven't yet created
//        // a coextensive key, create a snowy image.
//        if (new_user_key || m_hSnowyDIB == NULL)
//            MakeSnow(m_hOriginalDIB);
//
//        // Use the new settings, and sign the image.
//        Sign();
//        m_state = IMAGE_SIGNED;
//        if (((CDibLookApp *)AfxGetApp())->m_autoread)

```

```

(
    // Run the reader again to see if we recover message.
    Read(m_hSignedDIB, FALSE);
    m_state = IMAGE_SIGNED_AND_VERIFIED;
)

// Now see if a "signed image" view exists. If not, create it.
CreateUniqueView(SIGNED_VIEW);

// Now see if a "status image" view exists. If not, create it.
CDibView *p_statusView;
p_statusView = (CDibView *) CreateUniqueView(STATUS_VIEW);
EndWaitCursor();

// Refresh all of the views (Don't actually need to refresh Original one)
p_statusView->Resize();
UpdateAllViews(NULL);

// Some debug stuff related to checksums.
TRACE("signer checksum: %x\n", (int) m_pPackedMsg->GetSignerChecksum());
TRACE("Read checksum: %x\n", (int) m_pPackedMsg->GetReaderChecksum());
TRACE("Reader computed checksum: %x\n",
      (int) m_pPackedMsg->GetComputedReaderChecksum());
)

// CreateUniqueView()
// This function creates a new view of the indicated type, if and
// only if one does not already exist. It returns a pointer to
// the new view, if a new one is created, or a pointer to the
// pre-existing view of the specified type if one already exists.
// The "view type" argument is one of the view types from SignView.h,
// i.e. SIGNED_VIEW, ORIGINAL_VIEW, STATUS_VIEW
// View* CDibDoc::CreateUniqueView(int view_type)
//
// BOOL view found = FALSE;
// POSITION pos = GetFirstViewPosition();
// CView* pView;
// while (pos != NULL)
// {
//     pView = GetNextView(pos);
//
//     // If we find it, we return the pointer and we're done.
//     if ( (CDibView*)pView->GetViewType() == view_type )
//         return pView;
//
//     // The desired type of view doesn't exist, so we create it.
//     ChainFrame *mainFrame = (ChainFrame *) AfxGetApp()->m_pMainWnd;
//     mainFrame->MyOnWindowNew();
//
//     // Now find the newly created view (last in list) and set its type.
//     pos = GetFirstViewPosition();
//     while (pos != NULL)
//     {
//         pView = GetNextView(pos);
//         ((CDibView*)pView)->SetViewType(view_type);
//         return(pView);
//     }
//
//     ChangeViewType()
//
//     This function finds the view of the "old type", and changes its
//     type to "new type". If successful, it returns a pointer to
//     the newly changed view. If not, returns NULL.
//     The "view type" arguments are from the view types in SignView.h,
//     i.e. SIGNED_VIEW, ORIGINAL_VIEW, STATUS_VIEW, ALIGNED_VIEW
//     View* CDibDoc::ChangeViewType(int old_type, int new_type)
//     {
//         BOOL view found = FALSE;
//         POSITION pos = GetFirstViewPosition();
//         CView* pView;
//         while (pos != NULL)
//         {
//             pView = GetNextView(pos);
//
//             // If we find it, change its type, we return the pointer and we're done.
//             if ( (CDibView*)pView->GetViewType() == old_type )
//                 {
((CDibView*)pView)->SetViewType(new_type);
return pView;
}
// We get here only if we failed to find a view of "old_type"
return NULL;
}

// OnSettingsAutoPrint()
//
// When the user toggles the "Auto-print Report" item in
// the Options menu, this function is invoked. It simply
// toggles the corresponding member variable.
void CDibDoc::OnSettingsAutoPrint()
{
    if (m_autoprint == TRUE)
        m_autoprint = FALSE;
    else
        m_autoprint = TRUE;
}

// OnUpdatesSettingsAutoPrint()
//
// The framework calls this function whenever it is about
// to display the pulldown menu containing the AutoPrint
// Report option. Based on our internal state variable
// m_autoprint, we set or clear the check mark next to
// the menu item using the pCmdUI->SetCheck() function.
void CDibDoc::OnUpdatesSettingsAutoPrint(CCmdUI* pCmdUI)
{
    // Set or clear the check mark in the menu
    if (m_autoprint == TRUE)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

// OnSettingsReader()
//
// Invoked when the user selects the Controls->Reader...
// menu option. Presents a ReadParmsDlg dialog object, and
// deals with the operators inputs. On OK, the Read() function
// is called to use the current parameters and run the recog-
// nition core algorithms to try to detect an embedded
// digimarc message.
void CDibDoc::OnSettingsReader()
{
    ReadDlg dlg;
    rect;
    unsigned old_key;
    BOOL new_user_key = FALSE;
    int view_type;
    HDIB hImageToReadDIB;

    // Check to see if we are in a legal state for reading.
    if (m_state == NO_IMAGE)
    {
        MessageBox(NULL,
            "An 8 or 24 bit image must be loaded before using the Reader.",
            "Digimarc Signer Warning",
            MB_ICONINFORMATION | MB_OK);
        return;
    }

    // Determine the type of the active window
    view_type = GetActiveViewType();

    // If active window is not acceptable for reading, warn user & return
    if (view_type != ORIGINAL_VIEW &&
        view_type != SIGNED_VIEW &&
        view_type != ALIGNED_VIEW)
    {
        MessageBox(NULL,
            "The active window must contain an image to be read.",
            "Warning",
            MB_ICONINFORMATION | MB_OK);
        return;
    }

    // Set pointer to the image which is to be read.
    if (view_type == ORIGINAL_VIEW)

```



```

        hImageToReadDIB = m_hOriginalDIB;
    else if (view_type == SIGNED_VIEW)
        hImageToReadDIB = m_hSignedDIB;
    else if (view_type == ALIGNED_VIEW)
        hImageToReadDIB = m_pAlignedImage->GetHDIIB();
    else
    {
        MessageBox(NULL, "Bug in OnSettingsReader!", "Error", MB_OK);
        return;
    }

    // Initialize the dialog data
    dlg.m_user_key = m_pParams->GetKey();
    old_key = m_pParams->GetKey();
    dlg.m_msg_length = m_pParams->GetMessage().GetLength();
    dlg.m_gain = m_pParams->GetGain();
    dlg.m_bump_size = m_pParams->GetBumpSize();
    dlg.m_detail_lut_scale = m_pParams->GetLutScale();
    // dlg.m_use_super_reader = m_pParams->GetSuperReaderFlag();

    // Invoke the dialog box
    if (dlg.DoModal() == IDOK)
    {
        m_pParams->SetGain(dlg.m_gain);
        m_pParams->SetBumpSize(dlg.m_bump_size);
        m_pParams->SetLutScale(dlg.m_detail_lut_scale);
        // m_pParams->SetSuperReaderFlag(dlg.m_use_super_reader);

        // If signer has not yet been used, or length changes, need a msg.
        if (m_pParams->GetMessage().GetLength() != (int) dlg.m_msg_length)
        {
            // Create a dummy msg of all x's.
            CString dummy_msg = CString('x', dlg.m_msg_length);
            m_pParams->SetMessage(dummy_msg);
        }

        // Create a PackedMsg object w/ our dummy msg.
        if (m_pPackedMsg != NULL)
            delete m_pPackedMsg;
        m_pPackedMsg = new PackedMsg( (const char *) m_pParams->GetMessage());
        if (dlg.m_user_key != old_key)
        {
            m_pParams->SetKey(dlg.m_user_key);
            new_user_key = TRUE;
        }

        // This is going to take awhile
        BeginWaitCursor();

        // If the user seed has changed, or if we haven't yet created
        // a coextensive key, create a snowy image.
        if (new_user_key || m_hSnowyDIB == NULL)
            MakeSnow(hImageToReadDIB);

        // Run the reader and attempt to recover message, and compute metrics.
        Read(hImageToReadDIB, m_pParams->GetSuperReaderFlag());

        // Make the state transition: depends on which image was read.
        if (view_type == ORIGINAL_VIEW || view_type == ALIGNED_VIEW)
            m_state = SUSPECT_READ;
        else if (view_type == SIGNED_VIEW)
        {
            if (m_state != IMAGE_SIGNED_AND_SAVED)
                m_state = IMAGE_SIGNED_AND_UNVERIFIED;
        }

        // KLUDGE for debug. Need the signer timestamp set.
        WHY? 1/12/94
        m_pParams->UpdateSignTime();

        // Now see if a "status image" view exists. If not, create it.
        CDialogView_P_StatusView;
        P_StatusView = (CDialogView *) CreateUniqueView(STATUS_VIEW);
        EndWaitCursor();

        // Refresh all of the views (Don't actually need to refresh Original one)
        P_StatusView->DoResize();
        UpdateAllViews(NULL);

        // See if the checksum read and the checksum computed from the
        // read message string agree. If not, warn user.
        if (m_pPackedMsg->GetReaderChecksum() !=
            m_pPackedMsg->GetComputedReaderChecksum())
        {
            MessageBox(NULL,
                "The embedded checksum didn't match the computed checksum.",
                "Warning", MB_OK);
        }
    }

    // Find the active view, determine its type, and return
    // it to the caller. The type is one of those listed
    // in the Dlgview.h file.
    int CDialog::GetActiveViewType(void)
    {
        BOOL view_found = FALSE;
        POSITION pos = GetFirstViewPosition();
        CView* pView;
        while (pos != NULL)
        {
            pView = GetNextView(pos);

            // If we find it, we return the pointer and we're done.
            if ((CDialogView*)pView->IsActive() == TRUE)
                return ((CDialogView*)pView)->GetViewType();
        }

        // We can get here when other apps are running and Windows sends message
        // resulting in CDialog::OnUpdateFileSaveAs() being called.
        // MessageBox(NULL, "Error in GetActiveViewType!", "Error", MB_OK);
        return(UNKNOWN_VIEW);
    }

    // Return a pointer to the active view (i.e., a CDialogView*), or NULL
    // if something goes wrong.
    CDialogView * CDialog::GetActiveView(void)
    {
        BOOL view_found = FALSE;
        POSITION pos = GetFirstViewPosition();
        CView* pView;
        while (pos != NULL)
        {
            pView = GetNextView(pos);

            // If we find it, we return the pointer and we're done.
            if ((CDialogView*)pView->IsActive() == TRUE)
                return (CDialogView*)pView;
        }

        // We can get here when other apps are running and Windows sends message
        // resulting in CDialog::OnUpdateFileSaveAs() being called.
        // MessageBox(NULL, "Error in GetActiveViewType!", "Error", MB_OK);
        return(NULL);
    }

    // OnSettingsAutoread()
    // When the user toggles the "Auto-read after signing" item in
    // the Options menu, this function is invoked. It simply
    // toggles the corresponding member variable.
    // We currently also toggle the application level variable,
    // so that the settings are global to all docs.
    void CDialog::OnSettingsAutoread()
    {
        if (m_autoread == TRUE)
        {
            m_autoread = FALSE;
            ((CDialogApp *)AfxGetApp())->m_autoread = FALSE;
        }
        else
        {
            m_autoread = TRUE;
            ((CDialogApp *)AfxGetApp())->m_autoread = TRUE;
        }

        // OnUpdateSettingsAutoread()
        // The framework calls this function whenever it is about
        // to display the pulldown menu containing the Autoread
        // option. Based on our internal state variable

```

```

// m_autoread, we set or clear the check mark next to
// the menu item using the pCmdUI->SetCheck() function.
void CDibdoc::OnUpdateSettingsAutoread(CCmdUI* pCmdUI)
{
    // Set or clear the check mark in the menu
    if ((CDiblookApp *)AfxGetApp()->GetApp()->m_autoread == TRUE)
        pCmdUI->SetCheck(TRUE);
    else
        pCmdUI->SetCheck(FALSE);
}

// OnSettingAlign()
// This function is called when the user selects the "Align" menu option.
// A CFileDialog object is created and used in order for the operator
// to specify the name of the "Reference Image" (a signed or unsigned
// original image used as the template).
void CDibdoc::OnSettingAlign()
{
    CString refname;
    BOOL success_flag;

    // Create a filter for the types of files the file dialog will offer
    char szFilter[] =
        "Windows Bit Map Files (*.bmp)|*.bmp|Device Independent Bitmaps (*.dib)|*.dib|"
        "All Files (*.*)|*.*||";

    // Construct a file dialog
    CFileDialog
        fdDlg(TRUE,
            "",
            NULL,
            OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
            szFilter);

    // Over-ride the default title in the file dialog window
    fdDlg.m_ofn.lpstrTitle = "Select a template file to be used for alignment";

    // Display the file dialog
    if (fdDlg.DoModal() == IDOK)
    {
        // Get the name of the reference image file.
        refname = fdDlg.GetPathName();

        BeginWaitCursor();

        // Create an image object for the reference image.
        // If one already exists, delete it first.
        if (m_pRefImage != NULL)
            delete m_pRefImage;
        m_pRefImage = new Image(refname);

        if (m_pRefImage->GetFileOK == FALSE) // bail out if something went wrong
            return;

        // Display the reference image
        CreateUniqueView(RRF_VIEW);

        // UpdateAllViews(NULL);

        TRACE("Call the Align() function (this is a test of trace output.)\n");

        // Do the actual alignment and change update the state description.
        success_flag = Align_it();

        if (success_flag)
        {
            m_state = SUSPECT_ALIGNED;

            // Now, the template image object has had its packed data array replaced
            // by the aligned, co-extensive image. Need to move this packed data
            // into the DIB array for display (and possible file saving) purposes.
            m_pRefImage->UnpackData();

            // We now call the image the Aligned image, not reference
            m_pAlignedImage = m_pRefImage;
            m_pRefImage = NULL;

            CreateUniqueView(ALIGNED_VIEW);

            // Create a status view, if it doesn't already exist.
            CDibview *p_statusView;
            p_statusView = (CDibview *) CreateUniqueView(STATUS_VIEW);

            p_statusView->DoResize();

            UpdateAllViews(NULL);
        }
    }
}

// Align_it()
// This function is responsible for carrying out the alignment operation,
// by calling upon Geoff's core algorithms. It is assumed that on entry
// 1) m_hOriginalDib is DIB of the suspect image, already loaded.
// 2) m_pRefImage points to a image object with the template (or
// reference) image.
//
// BOOL CDibdoc::Align_it(void)
// {
//     int num_channels;
//
//     // Create an image object for the suspect image.
//     Image suspectImage(m_hOriginalDIB);
//
//     // Currently we require that the reference and suspect are of same type
//     // (i.e., both color or B&W).
//     if (suspectImage.GetBitsPerPixel() != m_pRefImage->GetBitsPerPixel())
//     {
//         MessageBox(NULL,
//             "Warning",
//             "The suspect and reference images must both be color or B&W",
//             return(FALSE);
//         MB_ICONINFORMATION | MB_OK);
//     }
//
//     // Construct Align object.
//     if (m_pAlign != NULL)
//         delete m_pAlign;
//
//     m_pAlign = new Align;
//
//     // Create the "byte-wise" packed data arrays from the DIB 4-byte packing
//     suspectImage.MakePackedData();
//     m_pRefImage->MakePackedData();
//
//     if (suspectImage.GetBitsPerPixel() == 8)
//         num_channels = 1; // B&W image
//     else if (suspectImage.GetBitsPerPixel() == 24)
//         num_channels = 3; // Color image
//
//     // Call the core algorithm to do the alignment.
//     m_pAlign->direct_registration(m_pRefImage->GetPackedData(),
//         m_pRefImage->GetXDIm(),
//         m_pRefImage->GetYDIm(),
//         suspectImage.GetPackedData(),
//         suspectImage.GetXDIm(),
//         suspectImage.GetYDIm(),
//         num_channels);
//
//     return(TRUE);
// }

// OnUpdateFileSaveAs()
//
// When the file pulldown menu is selected, this function is called
// upon to determine whether the "Save As..." menu item should be
// enabled. It determines the type of the current view, and if it
// is of a type for which we currently allow file saves, the menu
// item is enabled.
//
// void CDibdoc::OnUpdateFileSaveAs(CCmdUI* pCmdUI)
// {
//     int view_type;
//
//     // Determine the type of the current view.
//     view_type = GetActiveViewType();
//
//     // If the active view contains an image, we know how to save it.
//     if (view_type == ORIGINAL_VIEW ||
//         view_type == SIGNED_VIEW ||
//         view_type == ALIGNED_VIEW ||
//         view_type == STATUS_VIEW)
//     {
//         pCmdUI->Enable(TRUE);
//     }
//     else

```



```

// signer.cpp : Defines the class behaviors for the application.
//
#include "stdafx.h"
#include "signer.h"
#include "mainfrm.h"
#include "signdoc.h"
#include "signview.h"
#include "mychildw.h"

// #include "AFXPRIV.H"

#ifdef _DEBUG
#define THIS_FILE __FILE__
#endif

char *global_cmd_line_args;

// CDibLookApp

BEGIN_MESSAGE_MAP(CDibLookApp, CWinApp)
//{{AFX_MSG_MAP(CDibLookApp)
ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
//}}AFX_MSG_MAP
// Standard file based document commands
ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
// Standard print setup command
ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

// CDibLookApp construction
// Place all significant initialization in InitInstance

CDibLookApp::CDibLookApp()
{
    m_lpParams = NULL;
    m_autoread = FALSE;
}

CDibLookApp::~CDibLookApp()
{
    if (m_lpParams != NULL)
        delete m_lpParams;
}

// The one and only CDibLookApp object

CDibLookApp NEAR theApp;

// CDibLookApp initialization

BOOL CDibLookApp::InitInstance()
{
    // Standard initialization
    // If you are not using these features and wish to reduce the size
    // of your final executable, you should remove the following initialization
    // sections and adjust the corresponding values in the project
    // LoadStdProfileSettings(); // Load standard INI file options (including MRU)
    // Register document templates which serve as connection between
    // documents and views. Views are contained in the specified view
    AddDocTemplate(new CMultiDocTemplate(IDR_DIBTYPE,
        RUNTIME_CLASS(CDibDoc),
        RUNTIME_CLASS(CMyChildWnd), // I replace CMDIChildWnd
        RUNTIME_CLASS(CDibView)));

    // create main MDI frame window
    CMainFrame* pMainFrame = new CMainFrame;
    if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
        return FALSE;
    pMainFrame->ShowWindow(m_nCmdShow);
    pMainFrame->UpdateWindow();
    m_MainWnd = pMainFrame;

    // enable file manager drag/drop and DDE Execute open
    m_MainWnd->DragAcceptFiles();
    EnableShellOpen();
    RegisterShellFileTypes();

    // signer.h : main header file for the SIGNER application
}

// As a test, save a global copy of command line args
// global cmd line args = m_lpCmdLine;
// SIGNER application
// DEBUG: display the command line before we parse it.
// AfxMessageBox(m_lpCmdLine);

// simple command line parsing
if (m_lpParams->GetInputFileName() == NULL)
{
    // create a new (empty) document
    // OnFileNew();
}
else if ((m_lpCmdLine[0] == '-' || m_lpCmdLine[0] == '/') &&
(m_lpCmdLine[1] == 'e' || m_lpCmdLine[1] == 'g'))
{
    // program launched embedded - wait for DDE or OLE open
}
else
{
    // open an existing document
    OpenDocumentFile(m_lpParams->GetInputFilename());
}

// Try adding another window.
// pMainFrame->OnWindowNew(); // fails: this is a protected member.
// pMainFrame->SendMessage(ID_WINDOW_NEW);
// pMainFrame->MyOnWindowNewTest();

return TRUE;

// CaboutDlg dialog used for App About

class CaboutDlg : public CDialog
{
public:
    CaboutDlg() : CDialog(CAboutDlg::IDD)
    {
        //{{AFX_DATA_INIT(CAboutDlg)
        //}}AFX_DATA_INIT
    }

    // Dialog Data
    //{{AFX_DATA(CAboutDlg)
    enum { IDD = IDD_ABOUTBOX };
    //}}AFX_DATA

    // Implementation
protected:
    virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support
    //{{AFX_MSG(CAboutDlg)
    // No message handlers
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

void CaboutDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CAboutDlg)
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
    //{{AFX_MSG_MAP(CAboutDlg)
    // No message handlers
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

// App command to run the dialog
void CDibLookApp::OnAppAbout()
{
    CaboutDlg aboutDlg;
    aboutDlg.DoModal();
}

// CDibLookApp commands

// signer.h : main header file for the SIGNER application

```







```

CPP OBJ5=.\Release/
CPP_SBR5=.\Release/
# ADD BASE MTL /nologo /D "NDEBUG" /win32
# ADD MTL /nologo /D "NDEBUG" /win32
MTL PROJ=/nologo /D "NDEBUG" /win32
# ADD BASE RSC /I 0x409 /d "NDEBUG"
# ADD RSC /I 0x409 /d "NDEBUG"
RSC PROJ=/I 0x409 /fo "$(INTDIR)/signer.res" /d "NDEBUG"
BSC32=bscmake.exe
# ADD BASE BSC32 /nologo
# ADD BSC32 /nologo
BSC32_FLAGS=/nologo /o "$(OUTDIR)/SignerWin32.bsc"
BSC32_SBR5= \
    "$(INTDIR)/Mainfrm.sbr" \
    "$(INTDIR)/Sign.sbr" \
    "$(INTDIR)/Signdoc.sbr" \
    "$(INTDIR)/Coxkey.sbr" \
    "$(INTDIR)/Parmadlg.sbr" \
    "$(INTDIR)/Pft.sbr" \
    "$(INTDIR)/Stdafx.sbr" \
    "$(INTDIR)/Mychildw.sbr" \
    "$(INTDIR)/Packmsg.sbr" \
    "$(INTDIR)/Signview.sbr" \
    "$(INTDIR)/Image.sbr" \
    "$(INTDIR)/Param.sbr" \
    "$(INTDIR)/Signer.sbr" \
    "$(INTDIR)/Align.sbr" \
    "$(INTDIR)/Read.sbr" \
    "$(INTDIR)/Dibapi.sbr" \
    "$(INTDIR)/ReadDlg.sbr" \
    "$(OUTDIR)/SignerWin32.bsc" : "$(OUTDIR)" $(BSC32_SBR5)
$(BSC32) @<<
$(BSC32_FLAGS) $(BSC32_SBR5)
<<

LINK32=link.exe
# ADD BASE LINK32 oldnames.lib /nologo /stack:0x2800 /subsystem:windows /machine:IX86
# ADD LINK32 oldnames.lib /nologo /stack:0x4800 /subsystem:windows /machine:IX86
# SUBTRACT LINK32 /profile /debug
LINK32_FLAGS=oldnames.lib /nologo /stack:0x4800 /subsystem:windows \
/incremental:no /pdb:"$(OUTDIR)/SignerWin32.pdb" /machine:IX86 \
/def:"\signer.def" /out:"$(OUTDIR)/SignerWin32.exe"
D8P_FILE= \
    "\signer.def" \
    "$(INTDIR)/Params.obj" \
    "$(INTDIR)/Signer.obj" \
    "$(INTDIR)/Align.obj" \
    "$(INTDIR)/Read.obj" \
    "$(INTDIR)/Dibapi.obj" \
    "$(INTDIR)/ReadDlg.obj" \
    "$(INTDIR)/Mainfrm.obj" \
    "$(INTDIR)/Sign.obj" \
    "$(INTDIR)/Signdoc.obj" \
    "$(INTDIR)/Coxkey.obj" \
    "$(INTDIR)/Parmadlg.obj" \
    "$(INTDIR)/Pft.obj" \
    "$(INTDIR)/Stdafx.obj" \
    "$(INTDIR)/Mychildw.obj" \
    "$(INTDIR)/Packmsg.obj" \
    "$(INTDIR)/Signview.obj" \
    "$(INTDIR)/Myfile.obj" \
    "$(INTDIR)/Image.obj" \
    "$(INTDIR)/Signer.res" \
    "$(OUTDIR)/SignerWin32.exe" : "$(OUTDIR)" $(D8P_FILE) $(LINK32_OBJS)
$(LINK32) @<<
$(LINK32_FLAGS) $(LINK32_OBJS)
<<

!ELSEBT "$(CFG)" == "Signer - Win32 Debug"
# PROP BASE Use_MFC 1
# PROP BASE Use_Debug_Libraries 1
# PROP BASE Output_Dir "Debug"
# PROP BASE Intermediate_Dir "Debug"
# PROP BASE Target_Dir ""
# PROP Use_MFC 1
# PROP Use_Debug_Libraries 1
# PROP Output_Dir "Debug"
# PROP Intermediate_Dir "Debug"
# PROP Target_Dir ""
OUTDIR=.Debug
INTDIR=.Debug

ALL : "$(OUTDIR)\SignerWin32.exe" "$(OUTDIR)\SignerWin32.bsc"

CLEAN :

```











```

{
    TRACE("\tSelectPalette failed in CDibView::OnPaletteChanged\n");
}
}
return 0L;
}

// OnInitialUpdate()
// OnInitialUpdate()
// OnInitialUpdate()
void CDibView::OnInitialUpdate()
{
    CScreenView::OnInitialUpdate();
    ASSERT(GetDocument() != NULL);

    SetScrollSizes(WM_TEXT, GetDocument()->GetDocSize());
    // Resize this view's window based on the size of the image.
    ResizeParentToFit();
    GetParent()->SetWindowText(GetDocument()->GetTitle() + " - Original");
}

// OnActivateView()
// OnActivateView()
// OnActivateView()
void CDibView::OnActivateView(BOOL bActivate, CView* pActivateView,
                             CView* pDeactivateView)
{
    CScreenView::OnActivateView(bActivate, pActivateView, pDeactivateView);
    if (bActivate)
    {
        m_bThisViewActive = TRUE;
        ASSERT(pActivateView == this);
        OnDoRealize(WPARAM m_hWnd, 0); // same as SendMessage(WM_DOREALIZE);
    }
    else
    {
        m_bThisViewActive = FALSE;
    }
}

// OnEditCopy()
// OnEditCopy()
// OnEditCopy()
void CDibView::OnEditCopy()
{
    CDibDoc* pDoc = GetDocument();
    // Clean clipboard of contents, and copy the DIB.
    if (OpenClipboard())
    {
        BeginWaitCursor();
        EmptyClipboard();
        SetClipboardData(CF_DIB, CopyHandle((HANDLE) GetHBIT())); // pDoc->GetHBIT();
        CloseClipboard();
        EndWaitCursor();
    }
}

// OnUpdateEditCopy()
// OnUpdateEditCopy()
// OnUpdateEditCopy()
void CDibView::OnUpdateEditCopy(CCmdUI* pCmdUI)
{
    pCmdUI->Enable(GetHBIT() != NULL);
}

// OnEditPaste()
// OnEditPaste()
// OnEditPaste()
void CDibView::OnEditPaste()
{
    HBIT hNewDIB = NULL;
    if (OpenClipboard())
    {
        BeginWaitCursor();
        hNewDIB = (HBIT) CopyHandle(::GetClipboardData(CF_DIB));
        CloseClipboard();
        if (hNewDIB != NULL)

```

```

////////////////////////////////////
// SetViewType()
////////////////////////////////////
void CDibView::SetViewType(int type)
{
    CDibDoc* pDoc = GetDocument();
    switch (type)
    {
        case SIGNED_VIEW:
            m_viewType = SIGNED_VIEW;
            // Set the window title.
            GetParent()->SetWindowText(GetDocument()->GetTitle() + " -Signed");
            break;

        case REF_VIEW:
            m_viewType = REF_VIEW;
            // Set the window title.
            GetParent()->SetWindowText(GetDocument()->GetTitle() + " -Reference");
            break;

        case ALIGNED_VIEW:
            m_viewType = ALIGNED_VIEW;
            // Set the window title.
            GetParent()->SetWindowText(GetDocument()->GetTitle() + " -Aligned");
            break;

        case STATUS_VIEW:
            m_viewType = STATUS_VIEW;
            // Set the window title.
            GetParent()->SetWindowText(GetDocument()->GetTitle() + " -Status");
            break;

        default:
            // This is an error.
            // afxmessage
            break;
    }
}

////////////////////////////////////
// DisplayStatus()
////////////////////////////////////
void CDibView::DisplayStatus(CDC* pDC)
{
    CDibDoc* pDoc = GetDocument();
    TEXTMETRIC tm;
    CString text;
    CRect rect;
    CTime t;

    pDC->GetTextMetrics(tm);
    int col = 20*tm.tmAveCharWidth;
    int line = tm.tmHeight;
    ostrstream strm;
    createStatusStream(strm);

    int height;
    rect.top = 10;
    rect.left = 10;
    rect.right = 50 * tm.tmAveCharWidth;

    height = pDC->DrawText(strm.str(), -1, &rect, DT_EXPANDTABS | DT_CALCRECT);
    rect.bottom = height + 10;
    pDC->DrawText(strm.str(), -1, &rect, DT_EXPANDTABS);

    // Resize the scrollbars to fit the information it contains.
    CSize size = CSize(rect.right+10, rect.bottom);
    SetScrollSizes(MM_TEXT, size);
    if (m_boorezizeStatusView)
    {
        m_boorezizeStatusView = FALSE;
        ResizeStatusView(size);
    }

    // Once we call str(), we must delete the allocated space.
    delete strm.str();
    return;
}

////////////////////////////////////
// createStatusStream()
////////////////////////////////////
// This function creates a stream of characters in to the ostrstream passed in by
// parameter. The stream is described by the state argument. The state argument
// indicates our current program state, which influences what
// information is included in the stream data.
// void CDibView::createStatusStream(ostrstream &strm)
// {
//     CDibDoc* pDoc = GetDocument();
//     CTime t;
//     int state = pDoc->GetState();
//     PackedMag* pMag = pDoc->GetPackedMag();
//     strm << "\t\STATUS INFORMATION\n\n";
//     switch (state)
//     {
//         case NO_IMAGE:
//             // This case shouldn't come up - no menu access.
//             strm << "No image has been loaded.";
//             break;
//         case IMAGE_LOADED:
//             strm << "\tThe loaded image hasn't been signed or read.";
//             break;
//         case IMAGE_SIGNED:
//             strm << "IMAGE SIGNED AND VERIFIED.";
//             case IMAGE_SIGNED_AND_SAVED:
//             strm << "Signer Status\n\n";
//             strm << "\tOriginal Text:\t\t" << pMag->getAsciiMag() << "\n\n";
//             strm << "\tMessage Length:\t\t" << pMag->GetMagLength() << "\n\n";
//             strm << "\tGain Setting:\t\t" << pDoc->GetSignerParams()->GetGain() << "\n\n";
//             strm << "\tGamma:\t\t\t" << pDoc->GetSignerParams()->GetGamma() << "\n\n";
//             strm << "\tKey:\t\t\t" << pDoc->GetSignerParams()->GetKey() << "\n\n";
//             strm << "\tBump Size:\t\t" << pDoc->GetSignerParams()->GetBumpSize() << "\n\n";
//             strm << "\tDetail Gain:\t\t" << pDoc->GetSignerParams()->GetLutScale() << "\n\n";
//             strm << "\tChecksum:\t\t" << (unsigned) pMag->GetSignerChecksum() << "\n\n";

//             strm.fill('0');
//             t = pDoc->GetSignerParams()->GetTimeStamp();
//             strm << "\tTime of Signing:\t\t";

//             // Disable the 4270 warning. This is a bug in Microsoft's iomanip.h.
//             // Without this, the setw() io manipulator causes a warning.
//             #pragma warning(disable:4270)
//             strm << setw(2) << t.GetHour() << ':';
//             strm << setw(2) << t.GetMinute() << ':';
//             strm << setw(2) << t.GetSecond() << '.';
//             strm << setw(2) << t.GetMonth() << '/';
//             strm << setw(2) << t.GetDay() << '/';
//             strm << setw(2) << t.GetYear() - 1900;
//             strm << "\n\n";
//             // Reset fill character to default.
//             strm.fill(' ');

//             // Put the warning level back to the default.
//             #pragma warning(default:4270)

//             if (state == IMAGE_SIGNED AND SAVED)
//                 strm << "\tSigned image saved as:\t" << pDoc->GetFilename() << "\n\n";

//             if (state == IMAGE_SIGNED AND VERIFIED)
//             {
//                 strm << "Reader Status\n\n";
//                 strm << "\tRecognized Text:\t\t" << pMag->getRecoveredAsciiMag() << "\n\n";
//                 // Remove references to "super reader" for now
//                 if (pDoc->GetSignerParams()->GetSuperReaderFlag())
//                     strm << "\tAlternative Reader:\t\t" << "On" << "\n\n";
//                 else
//                     strm << "\tAlternative Reader:\t\t" << "Off" << "\n\n";
//                 // Adjust the floating point precision of the stream.
//                 strm.setf(16, ios::fixed, ios::floatfield);
//                 strm.precision(2);
//                 strm << "\tBit Success Rate (%) \t\t" << pMag->GetPercentCorrect() << "\n\n";
            }
        }
    }
}

```

```

// Print crude metric.
strm.precision(4);
strm << "\tBit Estimator Std. Dev.: \t" << pDoc->GetMetric() << "\n\n";

// Print range.
strm << "\tBit Estimator Range: \t" << pDoc->GetRange() << "\n\n";

strm << "\tEmbedded Checksum Read: \t" << (unsigned) pMsg->GetReaderChecksum()
<< "\n\n";

strm << "\tChecksum Calculated: \t" << (unsigned) pMsg->GetComputedReaderChecksum()
<< "\n\n";
}

break;

case SUSPECT_ALIGNED:
    AlignStatus = pDoc->GetAlignStatus(); // Get the align status

    strm << "Aligned Image Status\n\n";

    // Adjust the floating point precision of the stream.
    strm.setf(ios::fixed, ios::floatfield);
    strm.precision(2);

    strm << "\tRotation applied to suspect: \t" << a_stats.rotation << "\n\n";
    strm << "\tTranslation (X, Y): \t" << a_stats.x_trans
    << ", " << a_stats.y_trans << "\n\n";
    strm << "\tScaling (X, Y): \t" << a_stats.x_scale
    << ", " << a_stats.y_scale << "\n\n";
    strm << "\tRefinement: \t" << a_stats.refinement << "\n\n";

    break;

case SUSPECT_READ:
    strm << "Reader Status\n\n";

    strm << "\tAssumed Message Length: \t" << pMsg->GetMsgLength() << "\n\n";

    strm << "\tRecognized Text: \t" << pMsg->GetRecoveredAsciiMsg() << "\n\n";

    strm << "\tAssumed Key: \t" << pDoc->GetSignerParams()->GetKey() << "\n\n";

    strm << "\tBump Size: \t" << pDoc->GetSignerParams()->GetBumpSize() << "\n\n";

    strm << "\tDetail Gain: \t" << pDoc->GetSignerParams()->GetLutScale() << "\n\n";

    // Remove references to "super reader" for now
    //if (pDoc->GetSignerParams()->GetSuperReaderPlug())
    //    strm << "\tAlternative Reader: \t" << "On" << "\n\n";
    //else
    //    strm << "\tAlternative Reader: \t" << "Off" << "\n\n";

    // Adjust the floating point precision of the stream.
    strm.setf(ios::fixed, ios::floatfield);
    strm.precision(2);

    // Print crude metric.
    strm.precision(4);
    strm << "\tBit Estimator Std. Dev.: \t" << pDoc->GetMetric() << "\n\n";

    // Print range.
    strm << "\tBit Estimator Range: \t" << pDoc->GetRange() << "\n\n";

    strm << "\tEmbedded Checksum Read: \t" << (unsigned) pMsg->GetReaderChecksum()
    << "\n\n";

    strm << "\tChecksum Calculated: \t" << (unsigned) pMsg->GetComputedReaderChecksum()
    << "\n\n";

    break;
    default:
        break;
}

// Add a null terminator (DrawText needs it).
strm << '\0';

// ResizeStatusView()
// Resizes the status view frame window. The goal is to not
// move the upper left corner, and to not exceed the bounds of
// the MDI main frame window on the right or left borders.
void CDibView::ResizeStatusView(CSize status_size)
{
    const int bar_height = 27; // An empirically derived kludge

```







```

return(i);

// load_funky_lut()
// This function loads the scaling factor based on minimum linear funkiness
int load_funky_lut( float *funky_lut) // explicitly written for 8 bit
{
    float scale = (float)1.0;
    detail_start = 1;
    detail_stop = 50;
    length = (float)detail_stop - (float)detail_start;
    for(i=0; i<detail_start; i++) funky_lut[i] = (float)0.0;
    for(i=detail_start; i<detail_stop; i++)
    {
        funky_lut[i] = scale*((float)(i-detail_start)/length);
    }
    for(i=detail_stop; i<512; i++) funky_lut[i] = funky_lut[detail_stop-1];
    return(status);
}

// this function associates a given row and column value of a bump in the
// standard signature block with A) the bit plane of the message associated with the bump,
// output in the 'message bit_lut' variable array, and B) whether the '1' direction is up
// XOR lut=1, or down, XOR lut=0
// IMPORTANT: this also takes care of the basic XOR'ing operation between the message and
// the underlying code pattern (invert, don't invert)
int load_standard_message_block_lut(
    unsigned char *message, // if this is NULL, return the un XOR'ed array (for reading)
    long message_length,
    unsigned char *control_message, // this is the separate "always gotta be there" message
    long control_message_length, // its length
    short *message_bit_lut,
    unsigned char *XOR_lut,
    long read_or_write
){
    // this is a crude first version... April 1996
    // we're going with 16 control bits, and in this demo, we'll use all of them
    // to describe the raw message length as a short unsigned int
    //int *length_table = new int(15);
    //int *xblocks = new int(15);
    //int *yblocks = new int(15);
    int length_table[] = {16, 24, 32, 48, 64, 96, 128, 192, 256, 384, 512, 768, 1024, 1536, 3072};
    int xblocks[] = {8, 8, 4, 8, 4, 2, 4, 2, 2, 1, 2, 1, 1, 1, 1};
    int yblocks[] = {8, 8, 8, 8, 4, 8, 4, 2, 4, 2, 2, 2, 1, 2, 1};
    // find which length in the length table is next highest over current message_length
    long index=0;
    while( length_table[index] < message_length ){
        index++;
    }
    long xlength = (SIGNATURE_BLOCK_DIMENSION/2)/xblocks[index]; // length in bumps
    long ylength = (SIGNATURE_BLOCK_DIMENSION/2)/yblocks[index];
    long current_bit_kfoo, lfoo;
    long jump = SIGNATURE_BLOCK_DIMENSION;
    short actual_bit;
    long one;
    long i, j, k, l;
    short *message_bit;
    unsigned char *pxor;
    for(i=0; i<yblocks[index]; i++){
        current_bit = 1; // this is
        // simply a "mixing agent" so that given bit planes
        // don't congregate around edges (come up with a better way please please
        // the following uses the
        // 1 0
        // 0 1
        // formula of local bumps associated with a given bit plane, hence the 2's
        // floating around
        for(k=0; k<ylength; k++){
            // reset the pointers
            pmessage_bit = message_bit_lut[2*j*xlength + 2*(i*ylength+k)*jump];
        }
    }
    pxor = XOR_lut[2*j*xlength + 2*(i*ylength+k)*jump];
    kfoo = (k-6)*8;
    for(l=0; l<xlength; l++){
        foo = (l-6)*8;
        actual_bit = (short)(message_length + kfoo*4 + lfoo);
    }
    else { // this is the embedded data region
        actual_bit = (short)(current_bit & message_length);
        current_bit++;
        *pmessage_bit = *pmessage_bit+1; // *pmessage_bit+jump) =
        *pmessage_bit+jump+1 = actual_bit;
        pmessage_bit++;
    }
    if(read_or_write) one = 1;
    else {
        if(actual_bit >= message_length){
            if(control_message[actual_bit-message_length]) one = 1;
            else one = 0;
        }
        else {
            if(message[actual_bit]) one = 1;
            else one = 0;
        }
    }
    if(one){
        *pxor = 1;
        *pxor+1 = 0;
        *pxor+jump = 0;
        *pxor+jump+1 = 1;
    }
    else {
        *pxor = 0;
        *pxor+1 = 1;
        *pxor+jump = 1;
        *pxor+jump+1 = 0;
    }
    pxor++;
}
// delete [] length_table;
// delete [] xblocks;
// delete [] yblocks;
return(i);
}

int load_output_array(
    float *tweak,
    unsigned char *data_out,
    unsigned char *data,
    long xdim,
    long ydim,
    long bump_size,
    long jump_x
){
    unsigned char *pdata, *pdata_out;
    int i, j, k, temp;
    float ptweak, half = (float)0.5;
    pdata = data;
    ptweak = tweak;
    pdata_out = data_out;
    if(xdim == 1){ // single channel
        for(j=0; j<ydim; j++){
            temp = (int)((float)(pdata++) * (ptweak++) + half);
            if(temp==0) *pdata_out++ = 0;
            else if(temp>HIGHEST_GREY_VALUE) *pdata_out++ = HIGHEST_GREY_VALUE;
            else *pdata_out++ = (unsigned char)temp;
        }
    }
    else {
        for(i=0; i<bump_size; i++){
            ptweak = tweak;
            for(j=0; j<xdim; j++){
                temp = (int)((float)(pdata++) * (ptweak++) + half);
                if(temp==0) *pdata_out++ = 0;
                else if(temp>HIGHEST_GREY_VALUE) *pdata_out++ = HIGHEST_GREY_VALUE;
                else *pdata_out++ = (unsigned char)temp;
            }
            ptweak++;
        }
    }
}

```

```

unsigned char *data, // pointer to upper left corner of image block
long xdim, // absolute pixel dimension of current block
long original_xdim, // absolute pixel dimension of entire original image or passed array
long ydim, // absolute pixel dimension of current block
long original_ydim, // absolute pixel dimension of current block
long bump_size, // number of channels, e.g. 3 for RGB
long message_length, // message length
short *message_bit_lut, // this can be economized and reduced by 8 by using bitwise packing (I don't bother here)
float *detail_lut,
float *luminance_lut,
float *subliminal_grid,
float *subliminal_grid,
float *global_gain, // NULL if data is to be put back into input array
float *asymmetric_gain,
float *funky_lut
){
    long jump_x = Original_xdim - xdim; // this is the pointer offset for jumping rows
    unsigned char *pdata_out;
    long i,j;
    float *p1,*p2,*p3,*p4,*pbump,local_average_gain,detail_gain,diff;
    float *publiminal_grid,lum_gain,asym_gain,funky_gain,RGB;
    short *pbitt;
    unsigned char *pxor;
    double dtimp,bottomfunk;

    // set pdata_out based on (in place) versus new output array
    if (data_out == NULL){pdata_out = data;}
    else {pdata_out = data_out;}

    // calculate bitwies bias between original image, optionally degraded by common-model distortion), and each bit of the message, this will be used for differential gain of the bit planes to help "struggling" bits
    float *bit_bias = new float[message_length];
    for(i=0;i<message_length;i++){bit_bias[i]=(float)1.0;};
    // read block signature
    // convert_read_to_bias

    // dive into main loop
}

Main loop version 1 works in the following way. It is designed so that it can create a lagged version of the output in order to support either case of: A) where the input data array is replaced with the output array (in place), or B) where the data_out pointer is not null and is the actual output array.

--- THIS PARTICULAR VERSION EXPECTS CASE B ---

The main loop essentially operates bump by bump. It determines the local overall gain that should be applied to the given bump, then tweaks the individual pixels of the output bump and stores in the temporary array which is later written out into the ultimate output array.

// long xbumpdim = xdim/bump_size; // calling routine guaranteed this would never have a remainder
long ybumpdim = ydim/bump_size;
int *state_initial_bump_array;
int *xbumpsizes = xbumpdim; // adding '2' allows us to not worry about edges in core loops
float *bump0 = new float[xbumpsizes];
float *bump1 = new float[xbumpsizes];
float *bump2 = new float[xbumpsizes];
// load row 1 and row 2 (with row 0 data) for the first process step
// load bump_array should copy elements 0 and 1 with data bump 0
// and elements xbumpdim,data,xbumpdim+1 with data bump xbumpdim-1
load_bump_array(bump0,data,xbumpdim,zdim,bump_size,jump_x,1);
memcpy(bump2,bump1,xbumpsizes*sizeof(float));
// create tweak array for each raster of bumps
float *tweak = new float[xbumpdim];
float *ptweak;
float f1 = (float)1.0;
float f4 = (float)4.0;

for(i=0;i<ybumpdim;i++){
    // in order to avoid modulo housekeeping later on, copy the arrays downward
    // (as they are small too)
    memcpy(bump0,bump1,xbumpsizes*sizeof(float));
    memcpy(bump1,bump2,xbumpsizes*sizeof(float));
    if(i!=(ybumpdim-1)){ // load next bump row array
        load_bump_array(bump2,&data[(i+1)*bump_size*Original_xdim+zdim],xbumpdim,zdim,bump_size,jump_x,1);
    }
} else {} // leave bump2 alone

p1 = bump0+1;
p2 = bump1;
p3 = bump2+1;
p4 = bump1+2;
pbump = bump1+1;
publiminal_grid = &subliminal_grid((SIGNATURE_BLOCK_DIMENSION));
ptweak = tweak;
}

return(1);
}

core_sign_public_generation_1()
// problem has been reduced to basic block unit;
// the only special case is when xdim and/or ydim are not extended to full block size
int core_sign_public_generation1()

```

pbnt = kmessage bit\_lut(i\*SIGNATURE\_BLOCK\_DIMENSION);

pxor = xor\_lut(i\*SIGNATURE\_BLOCK\_DIMENSION);

for(j=0;j<xbumpdim;j++){ // this is the heart of the signing code and, BTW, csgg, ons, bump at a

Even this list above can get complicated in their inter-relationships and

significantly in our current lack of experimental data to support various

formulas.

/\* Here's the deal: (Written 4/26/96)

The goal of the signing process, beyond simply functioning, is to maximize the "numeric detectability" of an embedded signature while meeting some form of fixed "visibility/acceptability threshold" set by a given user/creator.

In service to design toward this goal, imagine the following three axis parameter space, where two of the axes are only half-axes (positive only), and the third is a full axis (both negative and positive). This set of axes define two of the usual eight octal spaces of euclidean 3-space. As things refine and "deservedly separable" parameters show up on the scene (such as "extended local visibility metrics"), then they can define their own (generally) half-axis and extend the following example beyond three dimensions.

The signing design goal becomes optimally assigning a "gain" to a local bump based on its coordinates in the above defined space, whilst keeping in mind the basic needs of doing the operations fast in real applications. To begin with, the three axes are the following. We'll call the two half axes x and y, while the full axis will be z.

The x axis represents the luminance of the singular bump. The basic idea is that you can squeeze a little more energy into bright regions as opposed to dim ones. It is important to note that when true "psycho-linear - device independent" luminance values (pixel DN's) come along, this axis might become superfluous, unless of course if the luminance value couples into the other operative axes (e.g. C\*xy). For now, this is here as much due to the sub-optimality of current quasi-linear luminance coding.

The y axis is the kitchen sink of "local hiding potential" of the neighborhood within which the bump finds itself. The basic idea is that flat regions have a low hiding potential since the eye can detect subtle changes in such regions, whereas complex textured regions have a high hiding potential. Long lines and long edges tend toward the lower hiding potential since "breaks and choplines" in nice smooth long lines are also somewhat visible, while shorter lines and edges, and mosaics thereof, tend toward the higher hiding potential. These latter notions of long and short are directly connected to processing time issues, as well to issues of the engineering resources needed to carefully quantify such parameters. Developing the working model of the y-axis will inevitably entail one part theory to one part picky-artist-empiricism. As the parts of the hodge-podge y-axis become better known, they can splinter off into their own independent axes if its worth it.

The z-axis is the "with or against the grain" axis which is the full axis - as opposed to the other two half-axes. The basic idea is that a given input bump has a pre-existing bias relative to whether one wishes to encode a '1' or a '0' at its location, which to some non-trivial extent is a function of the reading algorithms which will be employed, whose (bias) magnitude is semi-correlated to the "hiding potential" of the y-axis, and, ... fortunately ... can be used advantageously as a variable in determining what magnitude of a tweak value is assigned to the bump in question. The concomitant basic idea is that when a bump is already your friend, or even your friend in a big way, then why mess with it much, whereas when it is your enemy or a big time enemy, then you want to squash it like a four year old discovering how flat slugs can get underfoot. The really cool thing here is that, in general, the latter squashing operation tends more toward a local blurring operation as opposed to a local sharpening operation, and thus has somewhat less visibility per numeric tweak unit.

The above general description of the problem should suffice for many years. Clearly adding in chrominance issues will expand the definitions a bit, leading to a bit more signature bang for the visibility, and human visibility research which is applied to the problem of compression can equally be applied to this area but for diametrically opposed reasons. Fascinating possibilities truly. But alas, I am required to crank out some pot-shot first system which needs must neglect vast areas of the above general arenas. Here are its principles.

For speed's sake, local hiding potential will be calculated only based on a 3 by 3 neighborhood of pixels, the center one being signed and its eight neighbors. Beyond speed issues, there is also no data or coherent theory to support anything larger as well. The design issue boils down to canning the y-axis visibility thing, how to couple the luminance into this, and a little bit on the friend/enemy asymmetry thing. My guiding principles to start are simply to make a flat region, zero a classical pure maxima or minima region a "0" or the highest value, and to have "local lines", "smooth slopes", "saddle points" and whatnot fall out somewhere in between. In other words, let's pull out the darts and throw a few and see if any land on the board.

The following code has six basic parameters that will be used:

- 1) luminance
- 2) difference from local average
- 3) the asymmetry factor (with or against the grain)
- 4) minimum linear funkiness factor (out crude attempt at flat v. lines v. maxima)
- 5) bit plane bias factor

6) global gain (the user's single top level gain knob)

Even this list above can get complicated in their inter-relationships and

significantly in our current lack of experimental data to support various

formulas.

1) luminance is straightforward

2) difference from local average is also, and is rather important to our first generation stuff since it will directly be involved in reading signatures (assuming we don't get fancy phase-only reading algorithms going)

3) the asymmetry factor is a single scalar applied to the "against the grain" side of the difference axis of number 2 directly above, as well as being modified by the minimum linear funkiness factor below. (Certainly it can eventually become a function of other variables if and when data and theory supports such)

4) The minimum linear funkiness factor is admittedly crude but it should be of some service even in a 3 by 3 neighborhood setting. The idea is that true 2D local minima and maxima will be highly perturbed along each of the four

lines travelling through the center pixel of the 3 by 3 neighborhood while a visual line or edge will tend to flatten out at least one of the four linear profiles. (The four linear profiles are each 3 pixels in length, i.e., the top left pixel - center - bottom right; the top center - center - bottom center; the top right - center - bottom left; the right center - center - left center.) Let's choose some metric of "funkiness" or entropy as applied to three pixels in a row, perform this on all four linear profiles, then choose the minimum value for our ultimate parameter to be used as a "y-axis". Cheers to she or he who will take all of this to the next level of refinement.

5) The bit plane bias factor is an interesting creature with two facets. The pre-emptive plane and the post-emptive face. In the former, you simply "read" the unsigned image and see where all the biases fall out for all the bit planes, then simply boost the "global gain" of the bit planes which are, in total, going against your desired message, and leave the others alone or even slightly lower their gain. In the post-emptive modification you churn out the whole signing process replete with the pre-emptive bit plane bias and the other 5 parameters listed here, and then you e.g. run the signed image through heavy JPG compression AND model the "gastalt distortion" of line screen printing and subsequent scanning of the image, and then ... you read the image and find out which bit planes are struggling or even in error, ... you appropriately beef up the bit plane bias, and you run through the process again. If you have good data driving the beefing process you should only need to perform this step once, or you can easily Van-Citterize the process (arcane reference to a dithering process with some damping factor applied to the tweak).

6) Finally, there is the global gain. The goal is to make this single variable be the top level "intensity knob" that the slightly curious user can adjust if they want to. The very curious user can navigate down advanced menus to get their epifanatical hands on the other five variables here, and who knows what others in the future.

whew, that's the most commenting I've ever done. I must be getting old or maybe I'm just realizing it would be nice to leave a signpost or two in this first dart throwing.

```
// get luminance gain
lum_gain = luminance_lut( (int)*pbump );

// find current differential between bump value and local average
// this one can generally make use of inter-DN lut's;
// in this case, down to 0.25 of a DN
local_avg = *p1 + *p2 + *p3 + *p4;
diff = *pbump + f4 - local_avg;
detail_gain = detail_lut( (int)( fabs( (double)diff ) ) );
```

```
// now calculate tweak based first on message, include asymmetric gain
if( *(pXOR++) ){
    if(diff<0.0) asym_gain = asymmetric_gain;
    else asym_gain = f1;
    *ptweak = f1; // slip this one in here
}
else {
    if(diff>0.0) asym_gain = asymmetric_gain;
    else asym_gain = f1;
    *ptweak = -f1;
}
```

// funky time: minimum linear funkiness factor

```
// line 1
bottomfunk = fabs((double)( *pbump - *(p1-1))) + fabs((double)( *pbump - *(p3-1)));
// line 2
dtemp = fabs((double)( *pbump - *p1)) + fabs((double)( *pbump - *p3));
if(dtemp < bottomfunk) bottomfunk = dtemp;
```

```
// line 3
dtemp = fabs((double)( *pbump - *(p1-1))) + fabs((double)( *pbump - *(p3-1)));
if(dtemp < bottomfunk) bottomfunk = dtemp;
```

```
// line 4
dtemp = fabs((double)( *pbump - *p2)) + fabs((double)( *pbump - *p4));
if(dtemp < bottomfunk) bottomfunk = dtemp;
funky_gain = funky_lut( (int)bottomfunk );
```

```

// add in the bias
// *ptweak += bit_bias(*pbit++);

// now put them all together somehow, but how??
gain = global_gain * (lum_gain + asym_gain * (funky_gain + detail_gain));
*ptweak *= gain;

// then add in subliminal grid
// eventually make this subject to local gain as well
if (gain > GRID_MINIMUM_GAIN) *ptweak += *subliminal_grid;
subliminal_grid += *ptweak++; *pbump++; *p1++; *p2++; *p3++; *p4++;
}
load_output_array(*ptweak, *pdata_out, (i * bump_size * original_xdim * zdim),
                  *data[i * bump_size * original_xdim * zdim], xdim, zdim, bump_size, jump_x);
}

// optionally JPEG compress (or whatever compress) the output buffer
// find the new bit biases, fine tune the bit bias values and
// repeat the above operations

delete [] bit_bias;
delete [] bump0;
delete [] bump1;
delete [] bump2;
return (1);
}

// sign_public_generation_1()
// =====
int sign_public_generation_1()
{
    unsigned char *data;
    long xdim;
    long ydim;
    long zdim;
    long bump_size;

    // input data to be signed
    // it's x dimension
    // it's y dimension
    // generally 1 for B&W and 3 for 3x8bit RGB, data assumed R-G-B
    // number of pixels per singular bump along one dimension; e.g.2 for 2x2
    // bump size
    // either 0 or 1, inefficient but simple
    // length of message in bits, also length of message string
    unsigned char *message;
    unsigned char *control_message;
    long control_message_length;
    float *luminance_lut;
    float *detail_lut;
    float *subliminal_grid;
    unsigned char *data_out;
    // to be placed into input array 'data'
    float global_gain;
    float asymmetric_gain;

    long block_pixel_dimension, x_blocks, x_leftover, y_blocks, y_leftover, i, j, status=1;
    long temp_block_xdim, block_ydim;
    unsigned char *pdata, *pdata_out;

    block_pixel_dimension = SIGNATURE_BLOCK_DIMENSION * bump_size; // actual pixel dimension of a
    standard signature block
    x_blocks = 1 + (xdim-1)/block_pixel_dimension; // number of full (and possibly partial on the last)
    basic blocks
    x_leftover = xdim % block_pixel_dimension - xdim % bump_size; // ignore fractional bumps on ends
    y_blocks = 1 + (ydim-1)/block_pixel_dimension;
    y_leftover = ydim % block_pixel_dimension - ydim % bump_size; // ignore fractional bumps on ends
    // though the straggly bits on the ends can cause a bit of a bookkeeping issue, they save a lot of
    // headaches when it comes time to write simple core algorithms sans if statements

    // load the message length into the 16 bit long control message
    int ii = 1;
    control_message_length = 16;
    for (i=0; i<16; i++) {
        if (ii & (short)message_length) control_message[i] = 1;
        else control_message[i] = 0;
        ii *= 2;
    }

    // BE SURE TO COPY END FRACTIONAL BUMP DATA FROM INPUT TO OUTPUT, UNCHANGED
    // in other words, if xdim % bump_size or ydim % bump_size is non-zero, then we can
    // immediately copy the leftmost and bottommost strip into the output buffer, unchanged
    if (data_out != NULL) { // if data output buffer is the input buffer, no need for copying
        if (temp = (xdim % bump_size)) {
            for (i=0; i<ydim; i++) {
                pdata = *data + zdim * ((i+1) * xdim - temp);
                for (j=0; j<temp; j++) *pdata_out++ = *pdata++;
            }
        }
        if (temp = (ydim % bump_size)) {
            pdata = *data + (ydim - temp) * xdim * zdim;
            for (i=0; i<temp; i++) *pdata_out++ = *pdata++;
        }
    }
}

```